Consistent Global State:

Algorithms
and
an Application in
Distributed Garbage Collection

Shang Heping

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Science at
Concordia University
Montréal, Québec, Canada

August, 1988

ISBN 0-315-49054-3

Abstract

**Consistent Global State:
Algorithms
and an Application in
Distributed Garbage Collection**

Shang Heping

Consistent global state of a distributed system is an important paradigm. Efficient solutions to a number of distributed problems can be obtained using this fundamental concept. In this thesis, some problems of consistent global state detection are discussed extensively. Based on the degree of process coordination, consistent global state detection algorithms are classified. An experimental implementation of several detection algorithms is presented, and the performance of these algorithms is studied. A new type of consistent global state detection algorithm is proposed. The theme of this thesis is to study consistent global state detection in distributed systems, and to examine an application of a probabilistic detection algorithm.

TO My Wife Zou Xiaohui

And

My Parents

Acknowledgements

I would like to thank to my supervisor, Dr. H.F. Li, for his dedication to the topics herein presented; his persistent demand for high standards; his support in organizing and starting this paper in the first place; his financial assistance; and his encouragement.

I would also like to express my gratitude to those people who kindly assisted me in the course of my studies: K. Wang, X.L. Sun, K. Venkatesh, C. Lam, L. Lam, H.D. Cheng, D. Peters and P. Dubois. Credit is also due to the department of computer science, and particularly to Ms. S. Roberts whose cooperation has been much appreciated.

Thanks to Dr. J.W. Atwood, who proofread the typescript of this thesis. Special thanks to my friend R.N. Ponce, who cheered me up and helped me constantly.

How can I not mention my wife Zou Xiaohui, who devotes her life to me and our son, Junjun. Sincerely, thanks to you Xiaohui.

Table of Contents

List of Figures

Glossary of Symbols

$A \Longrightarrow B$	if A then B
$A \Longleftrightarrow B$	A if and only if B
$\sim A$	not A
$\forall$	for all
$\exists$	there exists
$x \in A$	x is an element of A
ΣA	summation all elements of A
$A \cup B$	A union of B
$A - B$	complement of B in A
$\alpha \longrightarrow \beta$	event α precedes event β
$LA < LB$	logical clock label LA earlier then logical clock label LB
$GA \ll GB$	global clock label GA earlier than global clock label GB
$TC_i(t)$	theoretical clock at place i
$RC_i(t)$	real clock at processes P_i
$LC_i(\alpha)$	logical clock at process P_i
$GC_i(\alpha)$	global clock at process P_i
CLA	Chandy's detection algorithm
VLR	Venkatesh's detection algorithm

Chapter I. Introduction

Development of microelectronics as well as interconnection and communication technologies have brought about the creation of Distributed Computer Systems (DCS) and their applications. In DCS, a number of independent and asynchronous processor-memory pairs are interconnected by communication channels. Inter-processor communication is achieved by passing messages through communication channels. Propagation of messages takes a finite but variable amount of time. There is no common clock running on DCS.

Many problems in distributed systems could be easily and efficiently solved if there were a mechanism which could take snapshots of the whole distributed system [Venkatesh88] [Natarajan86] [Elmargar86] [Chandy85]. However, the lack of a common clock, the variation of message transit time, the asynchronous characteristics among processors, and the absence of a master processor have made it impossible for any single process to capture an **Instantaneous State** of the entire system.

The concept of state for a complete system, which is intuitive in centralized systems, must be approached with caution in the DCS environment. Based on the requirements of applications, global states have been classified [Li87] into four main types, namely, **Statistical Global State**, **Stable Global State**, **Consistent Global State**, and **Synchronized Global State**. In this thesis, we will explore the consistent global state extensively.

The thesis is organized as follows: The following sections of this chapter are introductions to the **Models** of distributed computations that we will use, and some basic concepts and properties of consistent global state. The relationship among the physical time system, the virtual time system, and the consistent global state is examined in chapter two. Algorithms for detecting a consistent global state and a Classification Scheme of these algorithms based on the Degree of coordination of local state recording are presented in chapter three. A **Global State Kernel** (GSK) which carries a number of consistent global state detection algorithms is presented in chapter four. Heuristic test results of performance characteristics of some detection algorithms are also presented in this chapter. An **Application** of a probabilistic detection algorithm on garbage collection is the topic of chapter five. Finally, the conclusions of this thesis and some suggestions for future work are presented in chapter six.

1.1. Distributed Computer Systems

We are considering a class of multi-computer systems known as distributed computer systems (DCS). There is no universally accepted definition for a distributed computer system, but such a system can be characterized by the following properties [Davies81]:

- It includes an arbitrary number of computer systems and user processes;
- The architecture is modular, consisting of a possibly varying number of processing elements;
- Communication is achieved via message passing on shared communication structure (excluding shared memory);
- Some system-wide control is performed, so as to provide for dynamic inter-process cooperation and run time management;
- Inter-process message transit delay is variable and some non-zero time always exists between the occurrence of a send event at a source process and the corresponding occurrence of the receive event.

The distributed computations in this thesis are viewed as being embedded in a DCS environment. It is necessary to achieve a precise understanding of the DCS environment. The following initial assumptions are made regarding their operational characteristics.

Channel Assumptions:

- Inter-process communication is through point to point directed communication channels;
- Delivery of a message through the communication system takes a finite, variable, but positive time (Causality);
- an unbounded number of message buffers exists in each channel;
- No assumption is made on the order of message delivery in each channel. ■

Process Assumptions:

- Processes communicate via messages through explicit send and receive events;
- An event α in a process P is Atomic and changes the state of P from the previous state to a new state, which is denoted by $S(\alpha)$. An event can be a **Send Event** which puts a message into the buffer of an output channel of the process, or a Receive Event which gets a message from the buffer of an input channel of the process, or an Internal Event which invokes an internal operation of the process. An event is completely executed at one single process;
- A Common Clock does not exist in DCS. ■

1.2. Model of Distributed Computation

The main purpose of a model is to precisely define specific properties or characteristics of a system, so that the properties of the system can be analyzed or even proved. In this subsection, a model of distributed computations in a DCS environment is reviewed.

1.2.1. Partial Ordering

In the following, we define distributed computations in the DCS environment more precisely. We assume that a distributed computation is composed of a set of events that are distributed among a number of processes. Each event is Atomic and can be executed in a Single Process. A single process is represented by a Sequence of events that are ordered by the time of occurrence. Communication among processes is achieved by message passing.

As we know, a computation executed in a uni-processor system can be abstracted as a set of events and a precedence relation: In the distributed environment, however, it is sometimes impossible to say which of two events occurred first, i.e., the precedence relationship in a distributed computation is no longer totally ordered. The precedence relation of a distributed computation is a partially ordered relation [Lamport78] [Pratt86].

Definition 1.1.:

A distributed computation can be modelled as a set of partial ordered relations "$\longrightarrow$" on a set of events of a computation [Davies81]:

- If events α and β belong to a same process and event α Precedes event β, then $\alpha \longrightarrow \beta$;
- If event α is a send event and event β is the corresponding receive event, then $\alpha \longrightarrow \beta$ because of Causality;
- If $\alpha \longrightarrow \beta$ and $\beta \longrightarrow \tau$, then $\alpha \longrightarrow \tau$, i.e., the relation "$\longrightarrow$" is Transitive;
- Two distinct events α and β are Concurrent events if neither $\alpha \longrightarrow \beta$, nor $\beta \longrightarrow \alpha$;
- α $\tilde{\longrightarrow}$ α for any event α ($\tilde{}$ stands for Not), i.e., the relation "$\longrightarrow$" is Irreflexive. ∎

For any two events α and β, if $\alpha \longrightarrow \beta$, then we say event β Depends On α. If neither $\alpha \longrightarrow \beta$, nor $\beta \longrightarrow \alpha$, then we say events α and β are Independent.

1.2.2. Space-Time Model

Partially ordered events of a distributed computation can be graphically viewed as a Space-Time Diagram [Alford85] [Lamport78]. A space-time diagram is a two dimensional event graph as shown in Figure 1.1. Here the space represents processes P, Q and R in the vertical axis. The time represents the actual time of occurrence of each event in the horizontal axis. The nodes (points) denote events of the distributed computation. All the events that occur in a process belong to the same horizontal line. Message transmission is identified by a directed line from a send event to the corresponding receive event.

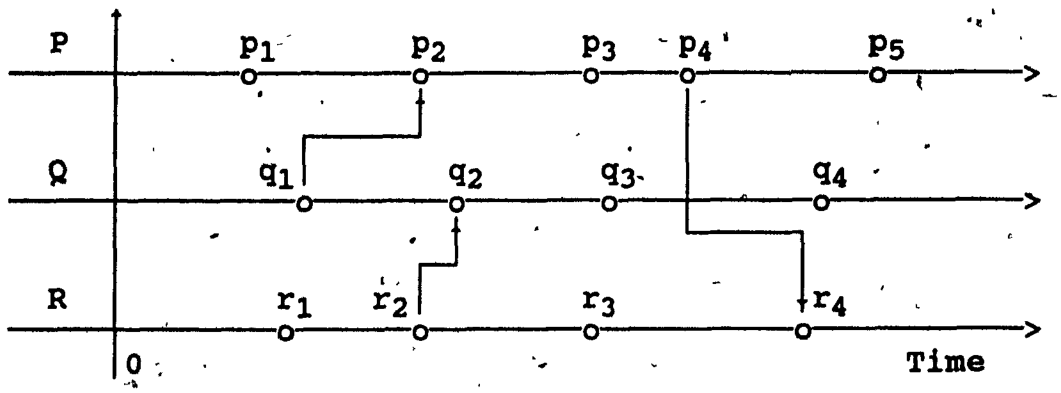


Figure 1.1. Space-Time Diagram

In a space-time diagram, events of different processes are partially ordered, due to the send-receive causality. It is easy to deduce that there is a path from event α to event β, if and only if α has a causally effect on β. If events are concurrent, then such a causal path does not exist between them in the space-time diagram. For example, in Figure 1.1, event r_4 depends on q_1. Therefore, there is a path from q_1 to r_4. Events p_3, q_3 and r_3 are independent. In general, a space-time diagram is an acyclic directed graph.

The space-time diagram is a graphical representation of partially ordered events of a distributed computation. It is used to Depict the (temporal) dependence of events, which represents a possible history of the execution of a distributed computation. The space-time diagram is an intuitive and visible method for describing physical realities. It can be used to define a global state of a distributed system. In this thesis we use the space-time diagram as a Synonym for a distributed computation.

1.3. Global State

A Global State of a distributed system is composed of the local states of all constituent processes and channels. The initial global state specifies that all the constituent processes are in their initial states and all the channels are empty. The global state is altered by the occurrence of a event in a constituent process. A Receive Event α can occur in the global state S_* if there is a message M at the head of the buffer of input channel C_j of process P_i. After the occurrence of event α, the global state of the system will change to S' which is the same as S_* except that the state of P_i changes to its next state $S(\alpha)$, and the state of C_j no longer includes M. Similarly, a Send Event can be envisaged. An Internal Event α occurs in the global state S_* when α changes the process state of P_i to $S(\alpha)$. After the occurrence of this event, the global state of the system will change to S' which is the same as S_* except that the process state of P_i changes to $S(\alpha)$.

Although an Instantaneous Global State of a distributed system is a mathematical fiction, it is a very useful fiction. The concept of global state in DCS is naturally based on the concept of instantaneous state. In fact the concept of global state in a DCS environment is an extension of the concept of system state in centralized systems. An instantaneous global state of a distributed system is composed of the local states of all its constituent processes and communication channels, which were recorded at some **Time Instant** [Li87]. An instantaneous global state can be depicted as a **Vertical Cut** in a space-time diagram. In Figure 1.2, line XX'=($S(p_2)$, $S(q_3)$, $S(r_2)$, Q-P(m_2), R-Q (m_3)) is an instantaneous global state recorded at t. Here $S(p_2)$, $S(q_3)$, and $S(r_2)$ are **Local Process States** corresponding to process P, Q, and R respectively. Q-P(m_2) and R-Q(m_3) are **Channel States** indicating a message M_2 in channel Q-P and a message M_3 in channel R-Q.

1.3.1. Space-Time Cut

We introduce two special nodes, a **Source Node S** and a **Terminal Node T**, in a space-time diagram. The two nodes form the first and last nodes in a space-time diagram as illustrated in Figure 1.3. A Cut in a space-time diagram is formed by a set of edges whose removal will disconnect the T node from the S node.

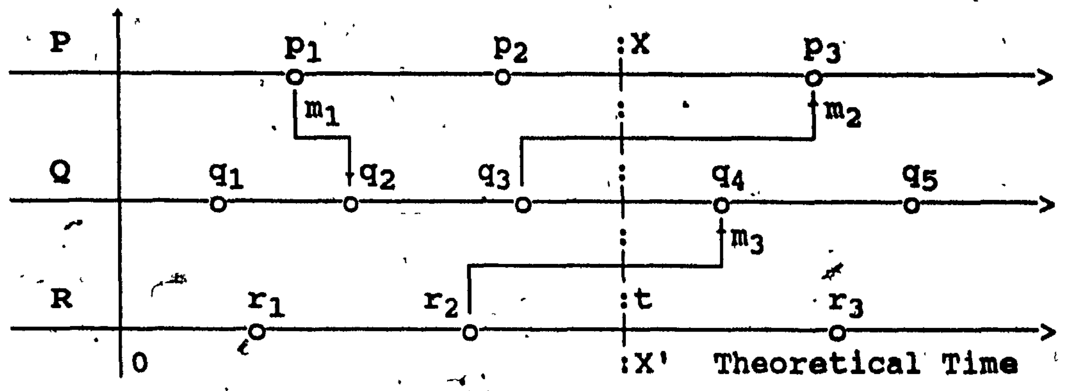


Figure 1.2. Instantaneous Cut

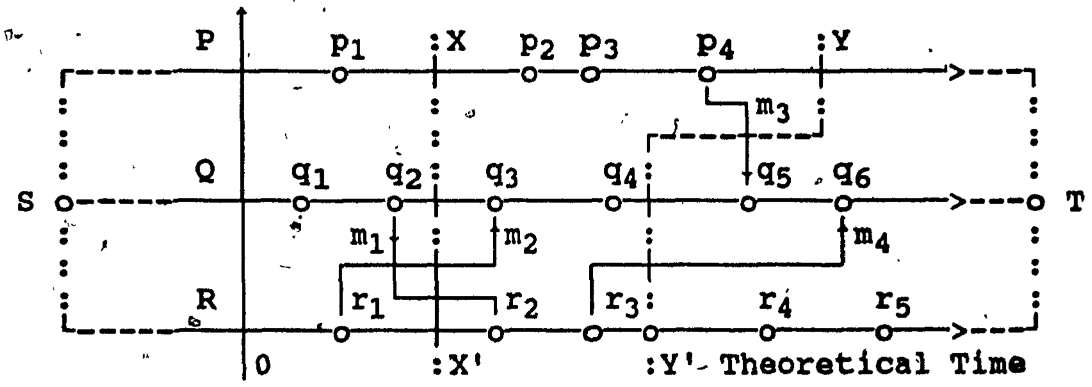


Figure 1.3. Space-Time Cut

Definitions 1.2. [Li87]:

- Edges along a horizontal line in a space-time diagram are Process Edges;

- Cross edges between two different horizontal lines are **Channel Edges**;
- A channel edge intersecting a cut is a **Forward Edge** if it emanates from an event node in the S-partition of the cut and terminates in the T-partition, else it is a **Backward Edge**;
- A cut is an **Instantaneous Cut** if it is a **Vertical Cut** in the space-time diagram at a particular time instant;
- A cut is a **Consistent Cut** if it does not contain backward channel edges and has precisely one edge on each horizontal line. ■

In Figure 1.3., cut XX'= (S(p_1), S(q_2), S(r_1), Q-R(m_1), R-Q(m_2)) is an instantaneous cut; and cut YY'=(S(p_4), S(q_4), S(r_3), P-Q(m_3), R-Q(m_4)) is a consistent cut.

1.3.2. Consistent Global State

Definitions 1.3.:

A consistent global state is defined as a recorded global state corresponding to a consistent cut. ■

As explained in [Chandy85] and [Li87], a recorded consistent global state S_* may not have actually occurred instantaneously in the course of the computation. However, if a consistent global state detection algorithm started with the computation in the initial state S_0 and terminated with the computation in the terminate state S_ϕ, then there exists a Sequence of computation (C1, C2), such that S_* is reachable from S_0 through C1 and S_ϕ is reachable from S_* through C2. This is known as the Reachability Property.

Another important property of the consistent global state is that a consistent global state preserves the Global Invariant property. A global invariant of a distributed computation, is a property satisfied by the processes as long as the Sequence of events of the computation possesses a Valid Serial Uni-processor Schedule. The state attained in the computation of the equivalent event trace is a consistent global state of the systems. In distributed computations, the actual physical time of occurrence of the events is not important, but the precedence relationship between matching send and receive events must be obeyed. The global invariant property can be used to reveal this precedence relationship.

Chapter II. Time Systems in DCS

The main reason for the impossibility of making use of the instantaneous global state in a distributed system is the absence of a Common Clock. If we had a common clock so that physical time could be maintained, then an instantaneous global state of the distributed system could be practically captured. Conceptually, the existence of a common clock enables one to establish a total ordering among the events of the distributed computation according to the time of occurrence. We could interpret this total ordering as a valid serial uni-processor schedule of the distributed computation. In a uni-processor system, an instantaneous snapshot of a computation can be obtained effectively. According to the definition in chapter one, this instantaneous global state is a consistent global state. In this chapter, we will explore the relationship between time and the consistent global state in the DCS environment.

2.1. Physical Time

The concept of time is fundamental to our way of thinking. In the absence of time, our world will become uninterpretable and nobody can cooperate with others and nobody needs to know others. When referring to time, most people are using Theoretical Time. Theoretical time is an abstraction of physical theory. In the real world, we maintain Real Time. Before defining the consistent global state by time, we first examine the difference between theoretical time and real time.

2.1.1. Theoretical Time and Clock

Theoretical time is a dimension of the physical universe which orders the sequence of events. It designates an event by an instant in this sequence [McCarthy87]. The passage of time is indicated by a device, called a clock.

Definition 2.1.:

Let $TC_i(t)$ denote the time obtained by reading a Theoretical Clock at place i at theoretical time t. The theoretical time system and clock satisfy the following two conditions:

TC1: $\forall_{i,t}\ [dTC_i(t)/dt = 1\ ]$

TC2: $\forall_{i,j,t}[TC_i(t) = TC_j(t)]$ ■

From TC1 and TC2, we observe that the theoretical time system has two functions: to order events and to assign time instants. A theoretical clock runs continuously forward and with a constant speed of unity (TC1). The reading of theoretical clocks at different locations is identical (TC2).

2.1.2. Real Time and Clock

The word Theory carries an unrealizability tone. Indeed, there are no clocks in the real world that satisfy TC1 and TC2. Clocks in the real world are **Real Clocks**. Real clocks are not perfectly accurate and are not perfectly synchronized.

Definition 2.2.:

Let $RC_i(t)$ denote the time obtained by reading a real clock at place i at theoretical time t. The real time system enforces the following two inequalities:

RC1: There exists a constant $\epsilon << 1$,

$$\forall_{i,t} \; [\,|dRC_i(t)/dt - 1| < \epsilon\,]$$

RC2: There exists a sufficiently small constant δ,

$$\forall_{i,j,t}[\,|RC_i(t)-RC_j(t)| < \delta\,] \quad \blacksquare$$

Obviously, TC1 and TC2 are theoretical abstractions of RC1 and RC2. The smaller the constants ϵ and δ are, the more similar are the behaviors of the two time systems. When ϵ and δ reach their limit zero, real clocks become theoretical clocks.

We introduce the concept of a **Common Clock**, which is a clock that does not precisely keep the theoretical time, but can be read identically anywhere (RC1 and TC2). With a common clock, all the events of a distributed computation can be totally ordered by their unique time instant of occurrence, and most concepts in centralized systems can be directly mapped into distributed systems. Also, a consistent global state can be easily captured by means of a common clock.

Theorem 2.1.:

An instantaneous global state of a distributed system obtained at a specific common time T is a consistent global state. Here, the local states of constituent processes and channels are recorded at a common clock instant T.

Proof: The instantaneous global state obtained at a specific time instant can be viewed as a vertical cut in a space-time diagram. According to our assumptions in chapter one, message delay in channels is strictly positive. Consequently, the channel edges are always forward going. Therefore, it is impossible for a vertical cut to contain a backward channel edge. Hence, an instantaneous global state recorded at a specific common time is a consistent global state. ■

Based on the notion of a common clock, mechanisms used for collecting system states in centralized systems can be easily extended to the DCS environment. We will come back to this type of detection algorithm in the next chapter.

Theorem 2.2.:

If each processes P_i maintains a real clock RC_i, and if the local process states and channel states of P_i in a global state G are recorded when $RC_i = T$, then the global state G may not be a consistent global state.

Proof: According to definition 2.2, real clocks in different processors may not be identical. For example, in Figure 2.1, RC_P reaches T after RC_Q reaches T. Cut XX'= $\{S(p_3), S(q_2), Q\text{-}P(m_2)\}$ contains a backward channel edge $Q\text{-}P(m_2)$. Consequently, XX' is not a consistent cut. ■

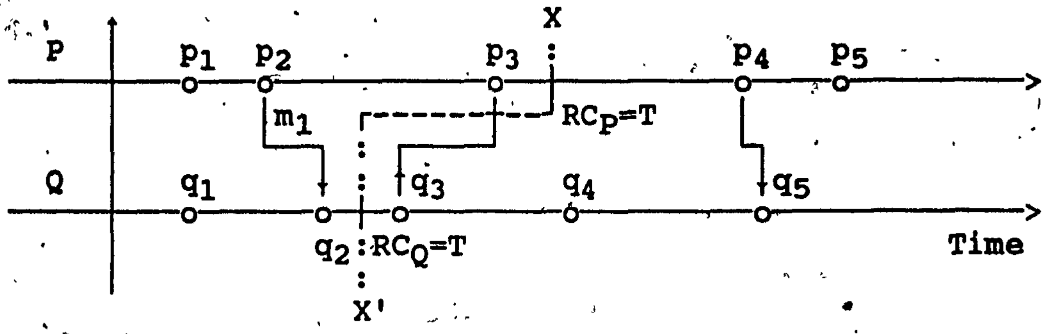


Figure 2.1. Real Time Instantaneous Cut

In Figure 2.1, process P has received a message yet to be sent at a future time. The global state information represented by this cut clearly violates **Causality.** Indeed, the events contained in the S-partition of the cut do not have a corresponding valid uni-processor schedule. A way to avoid this anomalous behaviour will be discussed in the next chapter.

The definition of consistent global state does not necessarily need the notion of common time. The common time is useful only because it makes the concept of global state in the DCS environment as intuitive as that in the centralized system. What is really needed ~~is~~ mechanisms which model the causality between arbitrary events. These mechanisms do not necessarily refer to theoretical time. We call a mechanism which completely characterizes the causality, but does not refer to theoretical time, as a **Virtual Time System**. In the following, we will develop two such mechanisms.

2.2. Virtual Time

In [Chandy85] and [Li87], the consistent global state of distributed computations is defined on a **Sequence** of events which corresponds to a valid serial uni-processor schedule. If event α precedes event β, then α appears before β in the sequence. The position of the events in a sequence can be regarded as the virtual time at which the events occurred. The virtual time system is a flexible abstraction of the real time system [Jefferson85] [Margan85][Lamport78]. It has been applied to convert partially ordered events of a distributed computation into a serial sequence. The theoretical time system, the common clock system and the virtual time system constitute the Time Systems in this thesis.

2.2.1. Logical Time and Clock

[Lamport78] introduced a Logical Time system to exploit the precedence relationships among distributed events. In this system, Logical Clocks are used for Labeling the events of a computation. The integer label of an event identifies the logical time at which the event occurs.

Definition 2.3.:

Let LC(α) denote the logical clock label of event α. The LC(α) must satisfy the following condition:

LC1: $\forall_{\alpha,\beta}[\alpha \longrightarrow \beta \Longrightarrow LC(\alpha)<LC(\beta)]$ ■

The partial order relation "$\longrightarrow$" is defined in Definition 1.1. LC1 asserts that if event α precedes event β in the sequence, then the logical clock label of α must be smaller than that of β; if the logical clock label of event α is smaller than that of event β, then the relation "β precedes α" is impossible.

Ordering events of a distributed computation according to their logical clock labels produces a valid uni-processor schedule of the computation. If the logical clock labels of two events α and β are identical (which is possible, we will discuss it later), then α can be placed ahead of β in the sequence, or vice versa.

Let $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ be a valid serial uni-processor schedule of a distributed computation according to their logical clock labels. Let S_i be a cut of the distributed computation with events $\alpha_1, \alpha_2, \ldots,$ and α_i in the S-partition, and events $\alpha_{i+1}, \ldots,$ and α_n in the T-partition.

Theorem 2.3.: S_i is a consistent cut.

Proof: Suppose there is a backward channel edge in the cut S_i, i.e., there exists $1 \le j \le i$, $i < k \le n$ such that $\alpha_k \longrightarrow \alpha_j$. Since $\alpha_k \longrightarrow \alpha_j$ implies $LC(\alpha_k) < LC(\alpha_j)$, the existence of a backward channel implies that event α_k should appear ahead of event α_j in the sequence. This is Contradiction as $k > j$. ■

The above theorem suggests that logical clocks can be used for detecting consistent global state. The details will be discussed in chapter three. The logical clock has been used in a number of applications, such as deadlock detection and termination detection [Lamport78] [Morgan85] [Chandy83].

The logical time system is only used for producing a sequence of events for a computation which possesses a valid serial uni-processor schedule. In the case of distributed computations, there are some events which are non causal and can be executed Independently of one another. Their logical clock labels cannot reveal such a relationship, which we call **Independence**.

Theorem 2.4.:

Independence among the events of a distributed computation cannot be revealed by examining the logical clock labels of the events.

Proof: Two events α and β in a distributed system are independent events if neither $\alpha \longrightarrow \beta$, nor $\beta \longrightarrow \alpha$. From LC1, if two events are concurrent, their logical clock labels do not have to satisfy any constraint. In other word, it could be that $LC(\alpha)>LC(\beta)$, $LC(\alpha)=LC(\beta)$, or $LC(\alpha)<LC(\beta)$. ■

Suppose we have a set of events of a distributed system and each event in the set is labelled with a logical time instant. Assume we have two labels $LC(\alpha)<LC(\beta)$. We cannot deduce whether α precedes β or they are independent. For example, in Figure 2.2, $LC(r_2)<LC(q_6)$ does not imply $r_2 \longrightarrow q_6$, while $LC(r_2)<LC(q_8)$ does imply $r_2 \longrightarrow q_8$. Therefore, we **Conclude** that the logical time system does not **Preserve** the independence information. We need a labelling mechanism which reveals independent distributed events.

A logical clock $LC(\alpha)$ is a function from a set of events of a computation to a set of integers. In practice, a $LC(\alpha)$ can be any function as long as $LC(\alpha)$ satisfies LC1.

The following algorithm is an implementation of the logical clock designed by [Morgan85]. Figure 2.2 shows an example of executing this algorithm.

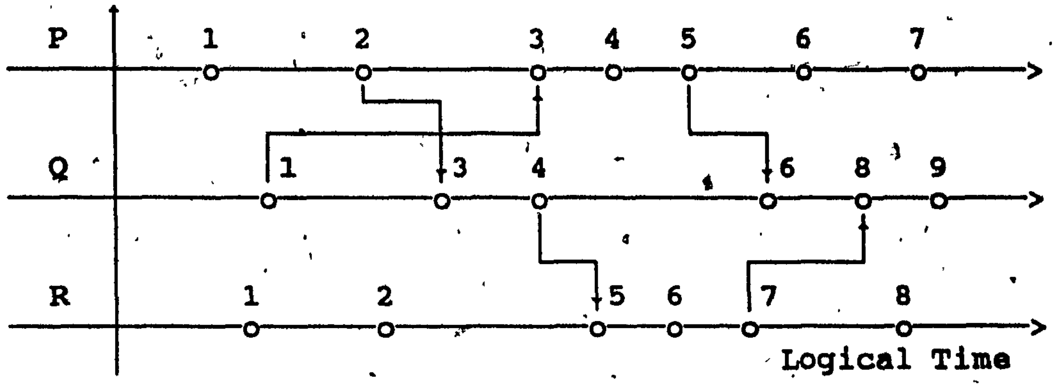


Figure 2.2. Logical Time Example

Algorithm 2.1.:

Each constituent process of a computation maintains an integer variable LC as its logical clock. The LC is initialized to zero. For each process, immediately before the occurrence of an event α, do:

```
Begin
  Case α is a Send Event (A);
     Begin
       LC := LC + 1;
       ST := LC;
       append ST as time stamp on the outgoing message;
     End;
```

```
    Case α is a Receive Event (B):
       Begin
         ST := time stamp on the income message;
         LC := 1 + Max( LC, ST);
       End;
    Case α is an Internal Event (C):
         LC := LC + 1;
    End;
  End;
```
■

When the event α occurs, the value of LC is regarded as the logical clock label of α.

Theorem 2.5.:

The labeling function $LC(\alpha)$ implemented by algorithm 2.1 satisfies LC1.

Proof: By definition 1.1, if event α_m depends on event α_0, then there exists a sequence of events $\alpha_0, \alpha_1, \ldots, \alpha_m$ so that events α_i $(0 \le i \le m-1)$ and α_{i+1} are either in the same process and α_i precedes α_{i+1}, or α_i is a send event and α_{i+1} is the corresponding receive event. In the former case, $LC(\alpha_i)<LC(\alpha_{i+1})$ by case (A) and (C) of algorithm 2.1, or in the latter case, $LC(\alpha_i)<LC(\alpha_{i+1})$ by case (B). The relation "<" is transitive, therefore, we have $LC(\alpha_0)<LC(\alpha_m)$. ■

2.2.2. Global Time and Clock

Since the publication of [Lamport78], the use of partial ordering as a logical clock in place of the common clock in a distributed system has become apparent. The events of a distributed computation are expressed as a partial order. We have observed that the logical time system explained previously implies a totally ordered relation and cannot be used to represent the independent events in a distributed computation. We therefore propose in this thesis a Global Time System to replace the logical time system. This global time system specifically addresses the independence of distributed events.

Like a logical clock, a global clock maintained in a global time system is a mechanism for labelling the events of a distributed computation. Suppose a distributed system is composed of N processes and a set of events S. The global clock label of an event represents the global time instant at which the event occurs.

Definition 2.4.:

A global clock label G is an N-element vector $<G_1, G_2, \ldots G_N>$. G_i $(1 \le i \le N)$ is an integer. The irreflexive and transitive relation « is defined on these global clock labels. Two labels LA and LB are related by "«" iff

$$\forall_{i \in N}[\ LA_i \le LB_i\ ] \quad \text{and} \quad \exists_{j \in N}[\ LA_j \ne LB_j\ ] \qquad \blacksquare$$

Definition 2.5.:

Let $GC(\alpha)$ denote the global clock label of an event α. $GC(\alpha)$ must satisfy the following requirement:

$$\text{GC1:} \quad \forall_{\alpha,\beta \in S}[\alpha \longrightarrow \beta \Longleftrightarrow GC(\alpha) \ll GC(\beta)] \qquad \blacksquare$$

GC1 asserts that event β depends on event α if and only if the global clock label of α is earlier ($\ll$) than that of β. Events α and β are independent if and only if neither the global clock label of α is earlier than that of β, nor vice versa.

The major difference of the global time and the logical time is that when events are labelled with global time, then we can discover the precedent relationship among them. However, with logical clock labels, we cannot distinguish two events are dependent or not.

A global clock $GC(\alpha)$ is a function from a set of events to a set of labels. The requirement which the function $GC(\alpha)$ must satisfy is contained in GC1. The following algorithm implements a $GC(\alpha)$. Figure 2.3 illustrates the algorithm.

Algorithm 2.2.:

Each constituent process of a computation maintains an N-element integer vector $GC = \langle C_1, C_2, \ldots, C_N \rangle$ as its global clock. The C_i are initialized to zero. For each process P_i, immediately before the occurrence of an event α, do:

```
Begin
   Case $\alpha$ is a Send Event (A):
      Begin
        GC := $\langle C_1, C_2, \ldots, C_i+1, \ldots, C_N \rangle$;
        ST := GC;
        append ST as time stamp on the outgoing message;
      End;
```

```
    Case $\alpha$ is a Receive Event (B):
       Begin
         ST := time stamp on the incoming message;
         for each j ($0 \le j \le N$) do
              $D_j$ := Max($C_j$, $ST_j$);
         GC := <$D_1$, $D_2$, ..., $D_i$+1, ..., $D_N$>;
       End;
    case $\beta$ is an Internal Event (C):
         GC := <$C_1$, $C_2$, ..., $C_i$+1, ..., $C_N$>;
    End;
 End;
```

When the event α occurs, the value of vector GC is the global clock label of α.

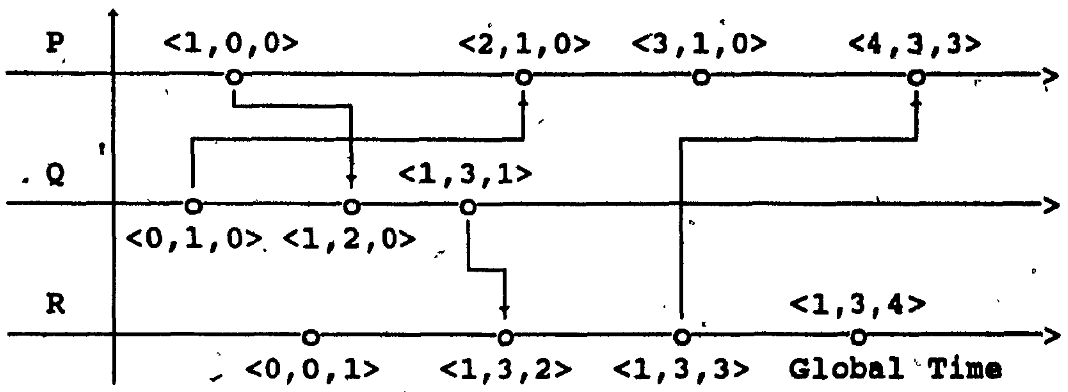


Figure 2.3. Global Time Example

Theorem 2.6.:

The function GC(α) implemented by algorithm 2.2 satisfies GC1.

Proof: For any two events $\alpha_0 \longrightarrow \alpha_m$:

By definition 1.1, event α_m depends on event α_0 implies that there exists a sequence of events α_0, α_1, ..., α_m so that events α_i and α_{i+1} ($1 \le i \le m-1$) are either in a same process and α_i precedes α_{i+1}, or α_i is a send event and α_{i+1} is the corresponding receive event. In the former case, $GC(\alpha_i) \ll GC(\alpha_{i+1})$ by case (A) and (C) of the algorithm 2.2. In the latter case, $GC(\alpha_i) \ll GC(\alpha_{i+1})$ by case (B). Relation "«" is transitive, therefore, we have $GC(\alpha_0) \ll GC(\alpha_m)$.

Conversely, for any two labels $GC(\alpha) \ll GC(\beta)$:

(1) suppose event $\beta \longrightarrow \alpha$. From the previous proof, we have $GC(\beta) \ll GC(\alpha)$ which is a Contradiction.

(2) Suppose events α and β are independent. Let α be in process P_i and β be in process P_j. According to our algorithm, $GC(\alpha)_i > GC(\beta)_i$ and $GC(\alpha)_j < GC(\beta)_j$, which means that $GC(\alpha) \ll GC(\beta)$ is Impossible. ■

Corollary:

The global clock $GC(\alpha)$ implemented by algorithm 2.2 is a One-To-One Function from an event set of a distributed computation to a global clock label set.

Algorithm 2.2 offers us a way to identify the events of a distributed computation uniquely. If every event of a distributed computation is stamped with its global time of occurrence, then we can easily reconstruct the space-time diagram of the distributed computation that took place. Since some properties, such as global state and global invariant can be visualized on a space-time diagram, the global time system is a useful tool for many applications.

The information maintained by the global time system is a superset of that maintained by the logical time system. For example, the global time system can be used to derive an event sequence which is a valid serial uni-processor schedule for a distributed computation. A sequence can be simply obtained by ordering ("«") the events linearly according to their global clock labels. In case neither GC(α)«GC(β), nor GC(β) « GC(α), we can arbitrarily place one ahead of the other.

The global time system is superior to the logical time system. The relation "«" is isomorphic to the relation "———>" among the events of a distributed computation, whereas the relation "<" on logical clock labels does not possess this property. With the global clock labels, we can identify whether two events of a computation are dependent. To identify independently distributed events is crucial in some applications. An example of such an application is presented in chapter five. The global time system is a practical model for representing distributed computations in the DCS environment.

Chapter III.

Consistent Global State Detection

Many distributed computations require gathering of global states in decision making. Such a requirement can be efficiently met if a good method for gathering global information exists. However, as explained in chapter one, an **Instantaneous Global State** of a distributed system cannot be realistically obtained. In other words, an exact snapshot of an entire distributed system is impossible to obtain.

Among the different kinds of global state classified in [Li87], the **Consistent Global State** is the most important one. A consistent global state of a distributed computation possesses the properties of **Causality** and **Global Invariant** preservability. These properties are useful in dynamic resource allocation, distributed system debugging and diagnosis, and fault-tolerant distributed computation [Li87]. However, consistent global states are not easily recordable. We must provide mechanisms to **Coordinate** individual processes in local state recording or to interpret and select relevant local state recordings and **Compile** them to form a consistent global state.

To detect a global state in distributed systems is not as easy as in centralized systems. In the DCS environment, a master site which controls all physical or logical resources may not exist. There is no common clock which can be used for process synchronization. What we can do is to allow individual processes to record their own process and channel states. This means that the global state detection algorithm is intrinsically distributed, i.e., every process contributes a little to the global state detection without perfect synchronization in physical time.

Typically, a global state detection algorithm consists of a set of processes that are coordinated in their local state recording. There is a Coordinator which assembles local states of individual processes. To detect a consistent global state, a Two-Phase Procedure is followed. In the first phase, called the Probe Phase, processes are coordinated in the process of local state recording. Because of the asynchronous characteristics of DCS, local states are not recorded simultaneously in different processors. Usually, the probe phase will guarantee that if an event is accounted for in one local recording, then any cause (predecessor) event of it must be also accounted for in some other process recording. In the second phase, called the Compilation Phase, the coordinator gathers all local states and compiles them to form a consistent global state [Venkatesh88]. Obviously the semantics of a consistent global state are application dependent.

A consistent global state detection algorithm should minimize the **Coordination Overhead**. This is not only because the cost of communication between computers is much higher than that within a computer, but also because of the severe delay to the pending computation caused by the communication delay if recording is coordinated.

In this chapter, we will explore and introduce some consistent global state detection algorithms in the DCS environment, hereafter referred to as detection algorithms. Based on the **Degree** of process coordination, we classify consistent global state detection algorithms into three categories: **Punctual Algorithms**, **Coordinated Algorithms**, and **Probabilistic Algorithms**. In most of our discussion, we concentrate on the probe phase of state recording. We will use the term local information to denote both process and its channel states.

3.1. Punctual Algorithm

There are various ways to achieve synchronization. A rather obvious way is to use **Physical Time**. In centralized systems, processes can be synchronized by using **Interruption**

Mechanisms. We borrow the idea of **Time-Out Interruption** in deriving a class of detection algorithms.

In a centralized system with a common clock, an instantaneous state can be captured by using a time-out mechanism. At a specific time instant, the time-out interrupt mechanism preempts the system control and enforces a process to take local state recording. We extend the time-out interrupt mechanism to DCS using one of the time systems presented earlier. This extended time-out mechanism enables us to force processes of a distributed computation to record their local information at an arbitrary Pre-specified time.

3.1.1. Using The Real Time System

To extend the time-out mechanism to DCS, a direct and simple way is to utilize the interruption system on each machine and synchronize their real clocks. In practice such a synchronization is inexact. As we proved in theorem 2.2, an instantaneous global state obtained by using real clocks in different processors may not be a consistent global state. However, if we can synchronize real clocks with a certain accuracy (defined below), then the anomalous phenomenon in state recording can be eliminated and a consistent global state can be captured by applying this extended time-out mechanism.

Let τ be the shortest time for transferring a message between two arbitrary processes. With τ and ϵ (RC1), we can synchronize real clocks in DCS so that the difference of any two clocks will be less than a small δ (RC2) [Lamport85] [Shin87].

Theorem 3.1.:

If a message of a distributed computation will not be put into the buffer of an input channel until its transmission time is longer than $\mu > \delta/(1-\epsilon)$ and if local states are recorded at the same real time T, then the instantaneous global state is a consistent global state.

Proof: The reason for the anomalous behaviour is that, in a global state, the real time instant $RC_j(t')$ of an effect event β at processor j is earlier than the real time instant $RC_i(t)$ of event α at processor i that causes β. To avoid this anomalous behaviour, we must guarantee that $RC_j(t')>RC_i(t)$. Since the time delay between a send event and the corresponding receive event is longer than μ, then we must show that:

$$\forall_{i,j,t}[RC_j(t+\mu) > RC_i(t)] \qquad \text{(A)}$$

From RC1: $\forall_{j,t}[|dRC_j(t)/dt - 1| < \epsilon]$ and $\epsilon \ll 1$

$\Longrightarrow$ (a) $\forall_{j,t}[RC_j(t)-RC_j(t+\mu) > (1-\epsilon)*\mu]$ (Omit)

(b) $\forall_{j,t}[RC_j(t+\mu)-RC_j(t) > (1-\epsilon)*\mu]$

$\Longrightarrow \forall_{i,j,t}[RC_j(t+\mu)-RC_i(t) > (1-\epsilon)*\mu-RC_i(t)+RC_j(t)]$

From RC2: $\forall_{i,j,t}[|RC_i(t) - RC_j(t)| < \delta]$

$\Longrightarrow \quad \forall_{i,j,t}[RC_j(t+\mu)-RC_i(t) > (1-\epsilon)*\mu-\delta]$

Inequality (A) will hold if $(1-\epsilon)-\delta > 0$, i.e.,

$\mu > \delta/(1-\epsilon)$ ■

The crucial problem here is to obtain a small enough δ (with certain τ and ϵ) so that a punctual algorithm using real clocks can be practically applied. Real clocks can drift with respect to one another. They must be periodically resynchronized. One simple scheme for synchronizing real clocks in DCS is to allow one computer to send a reset signal periodically to every other computer through a broadcasting medium. Another scheme is to adjust the rate of individual local clock periodically, based on information gathered by one computer [Gusetta87]. This has the advantage that there are no discontinuities in the value of time.

Using the time-out interrupt with the above restriction provides a simple and yet efficient detection algorithm. There is no coordination overhead in local state recording. The coordination overhead is replaced by the clock synchronization and the message delay caused by satisfying $\delta/(1-\epsilon)$. The disadvantages of this detection algorithm are that every message suffers a delay $\mu > \delta/(1-\epsilon)$, and computations at a process may be interrupted due to local state recording.

3.1.2. Using The Logical Time System

The logical time system is defined in definition 2.3. We have proved in theorem 2.3 that the logical time system can be used to detect a consistent global state in a DCS environment. The following is a detection algorithm:

Algorithm 3.1.:

Each constituent process of a computation maintains an integer variable LC as its logical clock. LC is initialized to zero. Each process also maintains an integer variable TO as its time-out clock. TO is set by the application. For each process P_i, immediately before the occurrence of an event α, do:

```
Begin
   OLD_LC := LC;

   case α is a Send Event:
      Begin
         LC := LC + 1;
         ST := LC;
         append ST as time stamp to the outgoing message;
      End;;
   case α is a Receive Event:
      Begin
         ST := time stamp on the incoming message;
         LC := 1 + Max( LC, ST);
      End;
   case α is an Internal Event:
         LC := LC + 1;
   End;

   if ( (OLD_LC < TO) && (LC ≥ TO) ) then
      Begin
         record local process state of Pi;
         record local channel states Pi;
         send local information to the coordinator;
      End;
End;
```

Theorem 3.2.:

A global state recorded at any particular logical clock label L by Algorithm 3.1 is a consistent global state.

Proof: Suppose a recorded global state is an inconsistent global state, then there must exist two events $\alpha \longrightarrow \beta$ so that β is accounted for in the global state while α is not included in the global state. From theorem 2.5, we have $LC(\alpha)<LC(\beta)<L$. Since β has been recorded in the global state, by algorithm 3.1, we have $LC(\beta)<L$, therefore $LC(\alpha)<L$. Thus, event α must be accounted for in the global state which is a Contradiction with the assumption. ∎

In order to record a consistent global state by means of a logical time system, processes should be interrupted right before their logical clocks reach a pre-specified label L, and then local states should be recorded. In a real time system, if a real clock i reaches T at time t_1 (refers to a clock k), and if a real clock j reaches T at time t_2 (refers to the same clock k), then $|t_1 - t_2|$ will be guaranteed to be less than a small δ (RC2). However, in the logical time system, if a logical clock i reaches L at time t_1 (refers to a real clock k), and if a logical clock j reaches L at time t_2 (refers to the same real clock k), then there will be no δ such that $|t_1 - t_2| < \delta$. This implies that even though the local states are recorded at the same logical clock label L, the physical time deviation of occurrence of local state recording may be unpredictably long. Figure 3.1 is an example of a poor case scenario for algorithm 3.1. In this example, we record a consistent global state at a logical clock label 4. Here, process R will not record its local state until the message from process P has been received, which is a long time after P has recorded its local state.

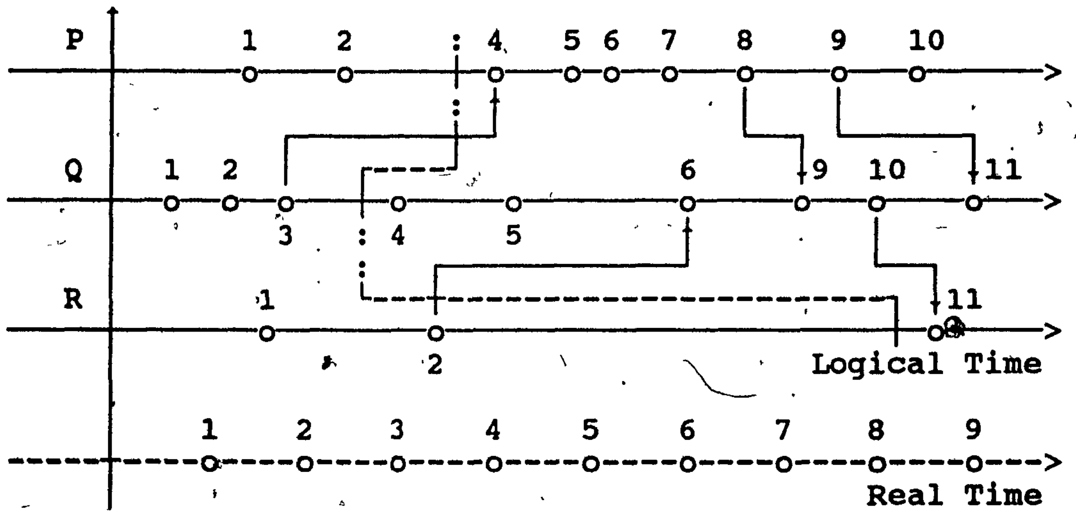


Figure 3.1. A Worse Case of Algorithm 3.1.

3.1.3. Using The Global Time System

The global time system can also be used to detect consistent global states. The following is a detection algorithm using the global time system:

Algorithm 3.2.:

Each constituent process of a computation maintains a N-element integer vector GC = $<C_1, C_2, \ldots, C_N>$ as its global clock. The C_i are initialized to zero. Each process also maintains an N-element integer vector TO = $<T_1, T_2, \ldots, T_N>$ as its time-out clock. TO is set by the application. For each process P_i, immediately before the occurrence of an event α, do:

```
Begin
   OLD_GC := GC;

   case α is a Send Event:
      Begin
        GC := <C1, C2, ..., Ci+1, Ci+1, ..., CN>;
        ST := GC;
        append ST as time stamp to the outgoing message;
      End;
   case α is a Receive Event:
      Begin
        ST := time stamp on the income message;
        for each i (0≤i≤N) do:
             Di := Max(Ci, STi);
        GC := <D1, D2, ..., Di+1, ..., DN>;
      End;
   case α is n Internal Event:
        GC := <C1, C2, ..., Ci+1, Ci+1, ..., CN>;
   End;

   if ( (OLD_GC « TO) && (TO « = LC) ) then
      Begin
        record local process state of Pi;
        record local channel states Pi;
        send the local information to the coordinator;
      End;
End;
```

Theorem 3.3.:

A global state recorded at a particular global clock label G by algorithm 3.2 is a consistent global state.

Proof: Suppose a recorded global state is an inconsistent global state, then there must exist two events $\alpha \longrightarrow \beta$ such that β is accounted for in the global state and α is not included in the global state. From theorem 2.6, we have $GC(\alpha) \ll GC(\beta) \ll G$. And from algorithm 3.2, we have $GC(\beta) \ll G$, therefore $GC(\alpha) \ll G$. Thus event α must be accounted for in the recorded global state. This is a Contradiction. ■

A global clock label $G = \langle T_1, T_2, \ldots, T_N \rangle$ at process P_i can be explained as the time immediately before the occurrence of its T_ith event or that immediately before the occurrence of the event effected by the T_jth event at process P_j ($1 \leq i, j \leq N$). As in the logical time system, the time difference between two local state recordings may still be very long.

3.2. Coordinated Algorithm

Implicitly, processes in punctual algorithms are coordinated by synchronized clocks. There is no other coordination overhead except in the form of clock synchronization. The punctual algorithms are easier to use than other algorithms. However, punctual algorithms require clock synchronization, and there may exist an unpredictable time delay among local state recording at different processes. Coordinated detection algorithms are used to overcome these disadvantages.

The main working characteristics of the coordinated algorithms discussed in this section are as follows:

- At least one of the processes of a distributed computation is designated as a Coordinator. Only a coordinator can acquire global states;
- A coordinator coordinates other processes to perform local state recording and pieces together all local information to form a global state;
- Processes cannot predict exactly when to record their local states.

A number of coordinated detection algorithms working under various environments has been proposed. As pointed out by [Gligor85], many distributed algorithms are erroneous. Among these proposals, [Chandy85] formalized the concept of consistent global state and presented a simple algorithm (CLA) to detect it correctly. [Li87] dealt with the concept of global state entirely and classified global states into four main types. [Venkates88] proposed an efficient detection algorithm (VLR) to work on more flexible environments.

Some other similar algorithms suggested by [Fische82] [Spzialett86] [Lai87] will not be discussed here.

3.2.1. Chandy's Algorithm [Chandy85]

CLA is an algorithm for detecting consistent global state. It is a simple algorithm in the sense that it requires communication channels to be **Lossless** and **FIFO**. As pointed out by [Venkatesh88], CLA is an inefficient detection algorithm because there is a **Coordination Message** to be transmitted on each channel. The following is a description of an extended CLA.

Algorithm[1] 3.3.:

Let P be a coordinator of a distributed computation. Each constituent process of the computation maintains a monotonic integer variable M, which indicates the ordinal number of the latest local state recording initiated by P. For each input channel I_j, there is an input message counter IC_j. M and IC_j are initialized to zero.

[1]This is an extended CLA algorithm. It supports multiple initiation of global state detections.

If P wants to detect a global state, do:

```
Begin
   $M := M + 1$;
   $PS_{pM}$ := local process state of P;
   for each input channel $I_j$ of P do
        $IC_{jM} := 0$;
   $CP_{sM}$ := empty;
   $ST_M := M$;
   send $ST_M$ to each of its output channels;
End;
```

For each process P_i, if it receives an M from I_j, do:

```
Begin
   $MC := ST_M$ received from the input channel $I_j$;
   $IC_j := IC_j + 1$;
   if ( $M < MC$ ) then
      Begin
        $M := MC$;
        $PS_{iM}$ := local process state of $P_i$;
        for each input channel $I_k$ of $P_i$ do
             $IC_{kM} := 0$;
        $CP_{iM}$ := empty;
        $ST_M := M$;
        send $ST_M$ to each of its output channels;
      End;
   $CS_{iM} := \{CS_{iM}\} \cup \{IC_{jM}\}$;
   if $ST_M$ received from every input channel of $P_i$ then
        send $PS_{iM}$ and $CS_{iM}$ to the coordinator;
End;
```

The probe phase of the M^{th} detection terminates when all processes have received ST_M from all input channels. The global state of the M^{th} detection is the local process states (PS_{sM}) and the channel states (CS_{sM}). ■

The coordinated local state recording procedure is performed at every process of the computation. So the recording architecture constitutes a layer between the application computation and the communication subsystem.

Coordination messages (ST_M) are not seen at the application layer.

The correctness of this extended CLA has been proved by [Venkatesh88]. Since a coordination message is received on every channel, the number of coordination messages for each global state detection with N processes and E channels will be equal to $O(E)=O(N^2)$.

3.2.2. Venkatesh's Algorithm [Venkatesh88]:

As pointed out by [Venkatesh88]:

- There are important applications of DCS such as distributed discrete event simulation and process control involving periodic sensor sampling which do not require the communication channels to be FIFO;
- It is impossible to achieve totally reliable communication;
- CLA cannot advantageously utilize either the "out-of-band" signalling or the broadcasting facilities in some interconnection networks.

He has proposed a coordinated detection algorithm VLR. VLR has been proved to function correctly even in Non-FIFO and/or Lossy communication channels.

The central idea behind VLR is that in order to determine the number of application messages in transit, we assign counters at both ends of a channel. This is unlike the case of CLA where counters are only associated with input channels. The following is a description of VLR.

Algorithm[2] 3.4.:

Let P be a coordinator of a distributed computation. Each constituent process of the computation maintains a monotonic integer variable M, which indicates the ordinal number of the latest local state recording initialized by P. For each input channel I_j and output channel O_k, there is an input message counter IC_j and an output message counter OC_k respectively. M, IC_j, and OC_k are initialized to zero.

If P wants to detect a global state, do:

```
Begin
    M := M + 1;
    $PS_{pM}$ := local process state of P;
    $CS_{pM}$ := $\forall_{j,k}\{ IC_j, OC_k \}$;
    ST := M;
    send ST to each of its output channels[3];
End;
```

[2]This VLR can work on Lossless and non-FIFO message channels. It supports multiple initiation of global state detections.

[3]To send the M along every output channel is only for improving detection response in case of losing an M. It is sufficient to send the M along the edges of a minimum spanning tree rooted at the process P.

For each process P_i, immediate before the occurrence of event α, do:

```
Begin
   case α is a Send Event:
      Begin
        let $O_k$ be the output channel for the message;
        $OC_k := OC_k + 1$;
        ST := M;
        stamp the outgoing message with ST;
      End;
   case α is a Receive Event:
      Begin
        let $I_j$ be the input channel of coming message;
        $IC_j := IC_j + 1$;
        RST := stamp on the incoming message;
        while ( RST > M ) do:
           Begin
             M := M + 1;
             $PS_{iM}$ := local process state of $P_i$;
             $CS_{iM} := \forall_{j,k}( IC_j, OC_k )$;
             ST := M;
             send ST to each of its output channels[3];
             send $PS_{iM}$ and $CS_{iM}$ to the coordinator;
           End;
      End;
   End;
End;
```

The probe phase of the M^{th} global state detection will be terminated when every process has recorded its local information which has been stamped with ST=M. The global state of the M^{th} detection is the local states (PS_{sM}) and the channel states (CS_{sM}). The channel state of channel C_j is $CS_{sM}(IC_j) - CS_{sM}(OC_k)$. ■

In VLR algorithm, each application message carries a flag to tolerate the possibility of non-FIFO delivery or loss of messages. The number of marker messages (M) in a system with N processes and E communication channels will be equal to $O(E)=O(N^2)$. Since a process does not have to wait for markers or flag messages from the other message channels after recording its own process state, the probe phase of the VLR algorithm is faster than that of the CLA algorithm. A performance study of VLR and CLA will be presented in chapter four.

If the communication subsystem of computations assures FIFO delivery and Lossless transmission, then a significant reduction of marker traffic can be obtained. In a FIFO and lossless environment, the marker messages only need to travel along the edges of a minimum spanning tree rooted at the coordinator. The number of marker messages in the new VLR algorithm working with FIFO and lossless communication channels will be (N-1), which has an improvement factor of O(N) over CLA. The VLR algorithm can be extended to detect other types of global state, which is out of the scope of this thesis and will not be examined here.

3.3. Probabilistic Algorithm

As we explained previously, the punctual detection algorithms assume synchronized clocks and the coordinated detection algorithms need coordination of processes in the process of recording. Both incur synchronization overhead. The response time of these algorithms may suffer because of this synchronization overhead. We therefore introduce probabilistic detection algorithms to overcome the above disadvantages. In this subsection, we develop such a probabilistic detection algorithm.

The central idea behind our probabilistic detection algorithm is to allow processes to perform their computation freely. There is no synchronization in either clock update or local state recording. The price we pay is the inability to guarantee consistency in every global state formed from the local recordings.

Let $S = (S(g_1), S(g_2), \ldots, S(g_N))$ be a global state of a distributed computation, g_i be the event of local state recording at process P_i, and $GC(g_i)$ be the global time instant at which process P_i recorded its local information $S(g_i)$.

Theorem 3.4.:

The global state $S = (S(g_1), S(g_2), \ldots, S(g_N))$ is a consistent global state, iff:

$$\neg\exists_{i,j \in n}[\ GC(g_i) \ll GC(g_j)\ ]$$

Proof: Suppose $GC(g_i) \ll GC(g_j)$ for some i and j. From definition 2.5, event g_i is a cause of event g_j. Hence there must exist a send event α and a corresponding receive event β, such that $GC(g_i) \ll GC(\alpha)$ or $g_i = \alpha$, and $GC(\beta) \ll GC(g_j)$ or $\beta = g_j$. The message sent by α is a backward channel edge. Therefore the recorded global state is not consistent.

Conversely, if a recorded global state S is consistent, the events g_1, g_2, ..., and g_N must be independent, i.e, there does not exist i, j ϵ N such that $GC(g_i) \ll GC(g_j)$. ■

Theorem 3.4 is the basis of our probabilistic algorithm. It is used to detect or construct a consistent global state from a set of local recordings of each process. Our probabilistic algorithm assumes the existence of the **Global Clock**. We assume the global clocks are maintained as a layer between the application and the communication subsystem.

In order to have an efficient probabilistic detection algorithm, we adopt the working scheme suggested by [Liskov86]. In this model, there is a **Logically Centralized Information Service** (LCIS) which stores a set of local information for each process. Any process can ask the LCIS for a consistent global state. The LCIS is responsible for gathering local information from individual processes and assembling the most recent local information of each process that can be used to form a consistent global state. The LCIS is not guaranteed to offer a consistent global state at any time, unless a large enough window or set of local recordings is maintained. The LCIS is to be superimposed on the underlying computation, i.e., it must run concurrently with, but not alter, the underlying application.

To detect a consistent global state, the probabilistic detection algorithm does not synchronize local information recording at each process. Instead, an individual process can record its local state at any time. If the communication subsystem supports message broadcasting, then before asking LCIS, a process can broadcast a signal to all other processes of the computation. Upon receiving the signal, a process can record its local information. This approach likely can enhance the probability of success and improve the response time of the detection algorithm.

The merits of the probabilistic algorithm are twofold. First, there is no coordination overhead: any process can request a consistent global state at any time, and processes work independently. Secondly, a consistent global state detection does not require synchronization which may delay the underlying computation. The probabilistic algorithm is more suitable for real-time applications.

Chapter IV. Global State Kernel

The Global State Kernel (GSK), an experimental implementation of the global state detection algorithms by [Chandy85] and [Venkatesh88], is presented in this chapter. The purpose of this work is to study the performance characteristics of the algorithms. The implementation and its measurement were performed on a **Distributed Network of SUN Workstations**, because measured data are believed to be more realistic than simulated results. We intend to compare the performance of these algorithms under some variations of DCS environments, and thus extract the influential factors of the environments.

4.1. GSK Implementation

The GSK provides a test bed for experimentation with different global state detection algorithms under various communication environments. Our GSK implementation follows the organization scheme discussed in chapter three. GSK is a software package that lies between the application and the communication system. It has several built-in algorithms for coordinated global state detection.

4.1.1. Environments and Requirements

The GSK was implemented on a group of SUN 3/50 workstations (12), under the **Network File System** (NFS). The NFS serves as a master of file management and inter-process communication control. Each SUN workstation contains a Motorola MC68020 processor, 4 megabytes random access memory, and a high speed 32 bit VME bus. The workstations communicate with one another through an Ethernet. There is no local disk in any workstation.

The Operating System is an enhanced 4.2BSD version of UNIX provided by SUN. There are two types of inter-process message channels available —— the **Lossy and non-FIFO** message channel and the **Reliable and FIFO** message channel. These channels are point to point channels. Each endpoint has a global name (Port). An inter-process message travels from an out endpoint (**Output Channel**) at a source process, via the NFS server, to the in endpoint (**Input Channel**) at the destination process. An output channel can communicate with one or more output channels, whereas an input channel can only listen to one input channel.

Five global state detection algorithms were implemented in the GSK. They are:

- Chandy's detection algorithm (CLA);

- Venkatesh's detection algorithm for reliable and FIFO communication channels (VLRc);
- Venkatesh's detection algorithm for lossy and non-FIFO communication channels (VLRc-non);
- Venkatesh's stable global state detection algorithm for reliable and FIFO communication channels (VLRs);
- Venkatesh's stable global state detection algorithm for lossy and non-FIFO communication channels (VLRs-non).

All above implemented algorithms supported multiple coordinators and simultaneous global state detection initiation. Since the stable global state is not the topic of this thesis, we will not discuss VLRs and VLRs-non. The main characteristics of the consistent global state detection algorithms are summarized in Figure 4.1.

4.1.2. GSK Design

In our implementation, we assumed that the GSK and the application processes are executed in the DCS environment, which were introduced in chapter one.

	Lossy & non-FIFO	Reliable & FIFO
CLA	1. CLA does not work under this channel. 2. The implemented CLA under this channel is the same as under the reliable & FIFO channel environment. 3. Measured data are accepted only when messages were correctly transferred.	Probe Phase 1. **Marker:** Travels in all communication channels 2. **Flag:** Not to be used. 3. **Termination:** Receive a marker from all of the input channels at each process site. Compilation Phase A channel state is the number of messages received between state recording and marker receiving.
VLR	Probe Phase 1. **Marker:** Travels along every channel. 2. **Flag:** Appended on all application messages. 3. **Termination:** All process receives a marker or flag. Compilation Phase A channel state is the difference of output & input message counters of the channel.	Probe Phase 1. **Marker:** Travel along minimum spanning tree rooted at coordinator. 2. **Flag:** Appended on all application messages. 3. **Termination:** All process receives a maker or flag. Compilation Phase A channel state is the difference of output & input message counters of the channel.

Notes:
(a) Listed properties are for one completed detection.
(b) Both Marker and Flag are coordinated messages, while marker is a standalone message and flag is attached on each application message.
(c) Local information of each process is send to the coordinator along a minimum spanning tree of the communication network rooted at the coordinator.
(d) The implemented CLA algorithm under lossy & non-FIFO channel is only for performance testing.

Figure 4.1. Algorithms and Environments

Generally speaking, the GSK is responsible for monitoring endpoints of each message channel, coordinating the application processes to record their local states, and assembling local information into an appropriate global state. In our implementation, the GSK is composed of a number of communication processes and signal handlers. For each application process there are two corresponding processes —— Input Channel Monitor (ICM), and Process Manager (PM). ICM is an input channel observer. Whenever there is a readable message in the message buffer of an input channel, the ICM will send a signal to its PM. The PM handles signals from both the ICM and the user. Specifically the PM performs the following operations:

- To append or extract coordination information (flags) on each application message;
- To generate special coordination message (markers);
- To analyze coordination information (markers or flags);
- To count the number of application messages at each endpoint of message channels;
- To enforce the local information recording at an appropriate time;
- To forward the recorded local information to the coordinator;
- To execute initiation procedures of global detection;
- To compile all recorded local information into a global state at the coordinator.

The coordinated detection algorithms assume that the topology of process communication is known. In order to handle dynamic changes of process topology, we assume processes are responsible to report changes in their channel connections. A General Manager is then established to relay all these changes to form the complete process topology. The responsibility of the GM is to gather and assemble the channel connection information from each PM, and to inform each PM of any change. The GM is a remote procedure. Any PM can call it without knowing where the GM located. Figure 4.2. shows the structure of the GSK.

The central idea behind CLA, VLRc, and VLRc-non is that local state recording is coordinated so that if an event is accounted for in the global state, then any event that may be a cause of that event should also be included. The number of user messages at each endpoint is counted in order to reveal the number of application messages in transit on the channels. If an application process sends a message, its PM will increment the associated output message counter; and upon receiving an application message, the PM will increment the corresponding input message counter. Local state recording is enforced by the PM. The PM, which serves as a coordinator, is responsible for initialling a global state detection, collecting all local information from each PM, and compiling it into a global state. Several coordinators can concurrently activate global state detection.

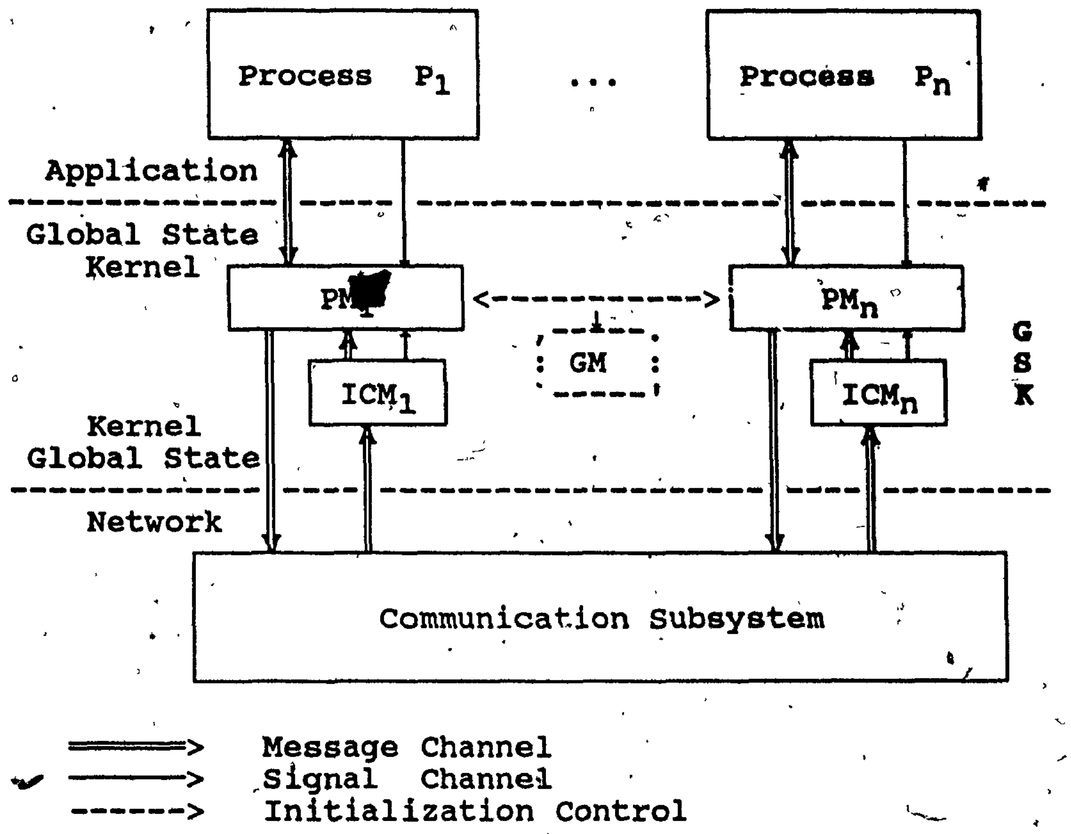


Figure 4.2. Global State Kernel

4.1.3. GSK Reference Manual

An application can invoke GSK functions by including the GSK library in their programs. GSK functions are written in C, which in turn invoke UNIX system functions (UNIX library (3s) and (2) [Sun86]). In this subsection we describe the syntax of the GSK functions. Most GSK functions have an error return, which normally is -1.

Descriptions:

Process Identifier (PID): A user must assign a unique name (PID) to each application process. The GSK identifies application processes by their PID.

Coordinator: Any application process can be designated to be a coordinator when the GSK is initialized. Only a coordinator can detect global states. There is no restriction on the number of coordinators in an application. However, the size of a coordination message is proportional to the number of coordinators, and will affect the performance of the detection algorithm.

Channel Topology (CT): Each application process must know, in advance, which processes it will communicate with. The channel connection is described in a file which has the following format:

```
from PID_1 to 11_PID, 12_PID, ...
from PID_2 to 21_PID, 22_PID, ...
                :
from PID_n to n1_PID, n2_PID, ... .
```

Notes that the file has a terminator period (".").

The channel topology description above specifies that there are message channels from process PID_1 to process 11_PID, from process PID_1 to process 12_PID, from process PID_2 to process 21_PID, etc.

Data Structure:

Process State Frame (PSF):

```
struct  PSF  {
        int  PID;
        char process_state[max_length];
        }
```

The constituents of a local process state depend on the application. The application should include a procedure that can record local states of each process. The procedure for local state recording should have the following form:

```
State_recorder( state)
        char *state
```

Channel State Frame (CSF):

```
struct  CSF  {
        int  from_PID[max_channels];
        int  to_PID[max_channels];
        int  counter[max_channels];
        }
```

The constituents of a channel state include the counter[] representing the number of messages in transit on the channel whose out endpoint is connected to process to_PID[] and whose endpoint is connected to process from_PID[].

Local State Frame (LSF):

```
struct  LSF  {
        int  coordinator_PID;
        int  first_MKNO, last_MKNO;
        PSF  process_state;
        CSF  channel_state;
        }
```

Global State Frame (GSF):

```
        struct  GSF  {
              int   INN;
              int   CFC;
              PSF   local_state[max_process];
              CSF   channel_state;
              }
```

The semantics of first_MKNO, last_MKNO, INN, and CFC will not be discussed in this thesis. Details are presented in [Li87] and [Venkatesh88].

Calling Sequences:

Before executing an application computation, the GM should be run at a console. The GSK is initialized when every application process has called function **start_GSK()**:

```
int  start_GSK( PID, Coordinator, C_topo)
           int      PID;
           char     Coordinator[];
           CT       C_topo;
```

If a process calls start_GSK() with coordinator="y", then it will be initialized as a coordinator.

```
int  collect_state( GSF)
     struct    GSF  *GSF;
     return    (GSF  *GSF);
```

Collect_state() is an asynchronous function, i.e., it will return to the caller without waiting for completion of global state detection. The GSF does not contain a valid global state until the field CFC of the global state frame (GSF) is set to -2.

After initialization, the GM will ask the user to identify the detection algorithm to be used, and then print the complete process topology on the console, which has the following format:

```
Total Application Processes: xxx;
 1. Pid: xxx;  Host Name: xxx;  Coordinator: yes|no;
 2. Pid: xxx;  Host Name: xxx;  Coordinator: yes|no;
               :
               :
Total Channels: xxx;
   1. From Pid: xxx ----> To Pid: xxx;
   2. From Pid: xxx ----> To Pid: xxx;
               :
               :
Minimum Spanning Tree For Sending Markers:
   1. Rooted At Coordinator Pid: xxx;
        1. From Pid: xxx ---> To Pid: xxx;
        2. From Pid: xxx ---> To Pid: xxx;
                    :
                    :
   2. Rooted At Coordinator Pid: xxx;
        1. From Pid: xxx ---> To Pid: xxx;
        2. From Pid: xxx ---> To Pid: xxx;
                    :
                    :
               :
Minimum Spanning Tree For Sending Local States:
   1. To Coordinator Pid: xxx;
        1. From Pid: xxx ---> To Pid: xxx;
        2. From Pid: xxx ---> To Pid: xxx;
                    :
                    :
   2. To Coordinator Pid: xxx;
        1. From Pid: xxx ---> To Pid: xxx;
        2. From Pid: xxx ---> To Pid: xxx;
                    :
                    :
               :
               :
```

All of the application messages actually filter through the GSK which supports the following communication primitives:

```
int   send_message( to_PID, length, message)
      int  to_PID, length;
      char *message;

int   receive_message( from_PID, message)
      int  from_PID;
      char *message;
      return  (int   length);
```

There are several functions which are used for interfacing with the communication subsystem. Since they are machine dependent, the details are not described here.

4.2. Performance Measurement

The performance analysis of an algorithm is quite important, because there are generally several candidates available for a given problem. Abstract complexity analysis often deals with the worst case scenario, and the conclusions are sometimes not convincing. In order to choose a good algorithm for an application, we still have to evaluate the performance and requirements of competing algorithms, through experimentation. In this subsection we present our measurement of the performance of the coordinated detection algorithms.

4.2.1. Diffused Computation

One of the applications that we have chosen for our performance study is the Diffused Computation [Dijkstra80]. A diffused computation consists of a number of diffused processes (N) and message channels (E). Diffused processes cooperate with one another by exchanging messages through point to point message channels. Among diffused processes, there is a special one, called the Initiator. At the beginning of a diffused computation, all diffused processes except the initiator are blocked, waiting for messages from any one of their input channels, which are initially empty.

A diffused computation is started by the initiator which sends messages to its output channels. After sending the first message, the initiator will behave the same as a normal diffused process. Upon receiving a message from an input channel, a diffused process can send an arbitrary (including none) number of messages on its output channels, and then block itself to wait for a message from any input channel. A diffused computation is terminated if all diffused processes are blocked, and there is no message in transit on any channel.

The global information of a diffused computation is collected with following purposes in mind:

(1) Global Invariant Validation: Let S_i ($1 \le i \le N$) be the number of messages that the diffused process P_i has sent out; let R_i be the number of messages that P_i has received. Let M_j ($1 \le j \le E$) be the number of messages in transit at the channel C_j. Obviously, the following

equation will be always hold during the diffused computation:

$$\Sigma S_i = \Sigma R_i + \Sigma M_j$$ L

The above global invariant is used to validate our GSK implementation. The constituents of the process state of process P_i are S_i, R_i, and M_i.

(2) **Termination Detection:** If each diffused process sends only a finite number of messages, then the computation will terminate eventually. For termination detection, the constituent of a local process state for P_i is its working state. The working state of a diffusion process is either Blocked or Waiting.

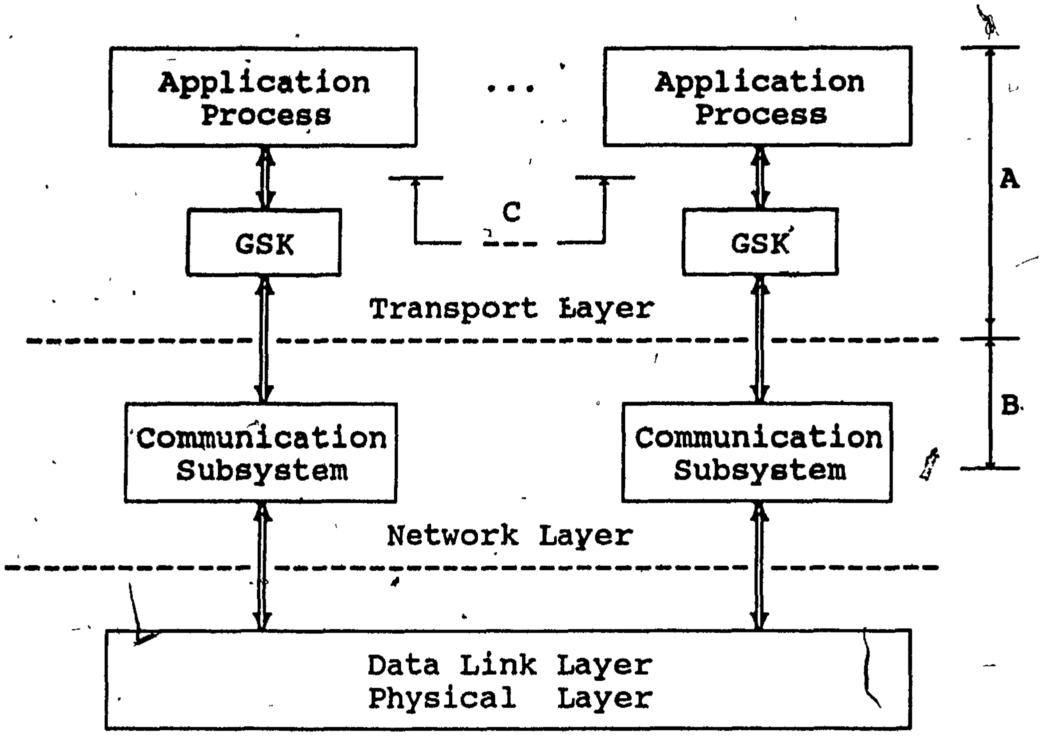
The reasons for applying the diffused computations for our evaluation are:

- The diffused computation is simple;
- It does not place any restriction on the process topology, which enables us test some best and worst cases for the detection algorithms;
- It does not place any restriction on the process communication. We can easily control the message broadcast pattern on any channel;
- It takes little CPU time for its own computation.

4.2.2. Communication Subsystem

In conventional distributed systems, the inter-process communication is expensive compared with other system functions. When distributed algorithms are implemented in a DCS environment, the inter-process communication overhead becomes critical. In the GSK, the communication overhead is due to the marker traffic. Furthermore, the real message delay is very difficult to characterize accurately enough by theoretical analysis. In this subsection, we will study the performance characteristics of our communication system. We will pay special attention to the performance behaviour of the different types of channels.

The SUN Ethernet local network uses the **Internet Protocol** [SUN86]. It allows processes at different workstations to communicate with one another by point to point channels, called **Sockets**. A socket has a type which specifies the semantics of the communication. A **Stream** type provides sequenced, reliable, and two-way connection based on byte streams with an out-of-band data transmission. A **Dgram** type supports datagrams, which are connectionless, non-sequenced, and unreliable, and have a fixed maximum message length. A **RAW** socket provides access to internal network interfaces. The Raw socket is available to super-users only. System function **Getusage()** is used to obtain information about any resource consumed by the processes. Figure 4.3 shows the structure of the SUN local network.



A: User Time --- Time spent in user processes
B: System Time --- Time spent in system processes
C: Real Time --- Time spent to transmit of a message

Figure 4.3. Inter Process Communication

We suspect that when the communication system saturates, message propagation time will increase dramatically. Also as shown in Figure 4.3, the bandwidth of physical cables and the communication subsystem may affect the performance characteristics of message transmission. In order to identify the bottleneck of our communication system, we isolate these factors from one another, and test each one independently. Testing processes are described below.

A Test Process was designed to measure the performance of the communication system. The test process sends a message of 260 bytes to an Echo Process located at another workstation. The test process keeps track of the local time delay from the sending of a message to its return from the echo process. Each type of channel is evaluated under the following three situations:

(a) Only the test and echo process-pair is running in the system. We denote such a situation by **Communication System Idle;**

(b) 30 **Disturbance** Process-pairs, which are evenly distributed among the other 10 workstations, execute concurrently with the test and echo process-pair. A disturbance process-pair does nothing but repeatedly send long messages to one another. This test is designed to saturate the physical and data link layer of the communication network. This situation is described as **Cable Busy.**

(c) Three disturbance process-pairs executed concurrently with the test and echo process-pair on the same workstations. These disturbance process-pairs are given a lower priority than the test and echo processes. This designed to saturate the communication subsystem. We call such a situation as **Subsystem Busy.**

Figure 4.4 shows the message propagation delay for Stream channel from 3:00 am to 9:00 am. Each sampled data is an average of 200 tests. Figure 4.5 shows the message propagation delay for Dgram channel. Figure 4.6 is the average of our test. Figure 4.7 shows the performance comparison of the two different types of channels under the different test situations.

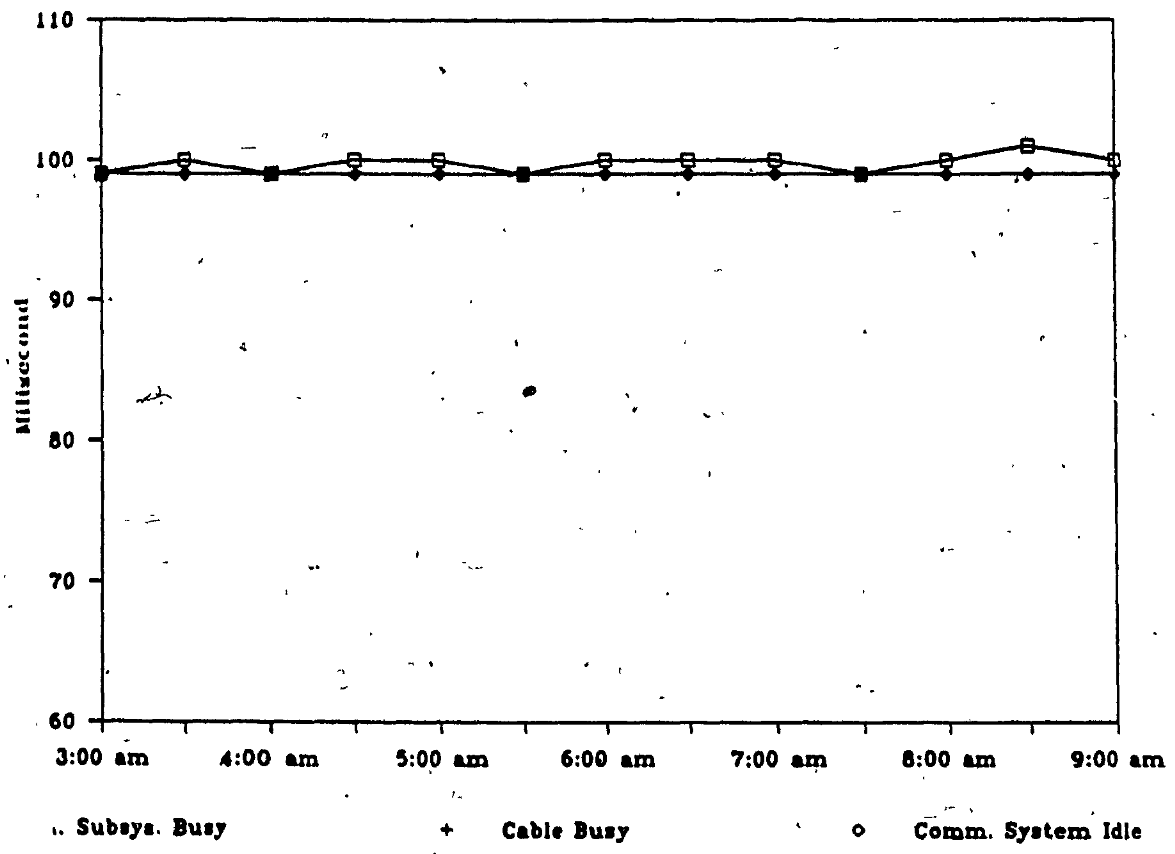


Figure 4.4. Stream Channel Propagation

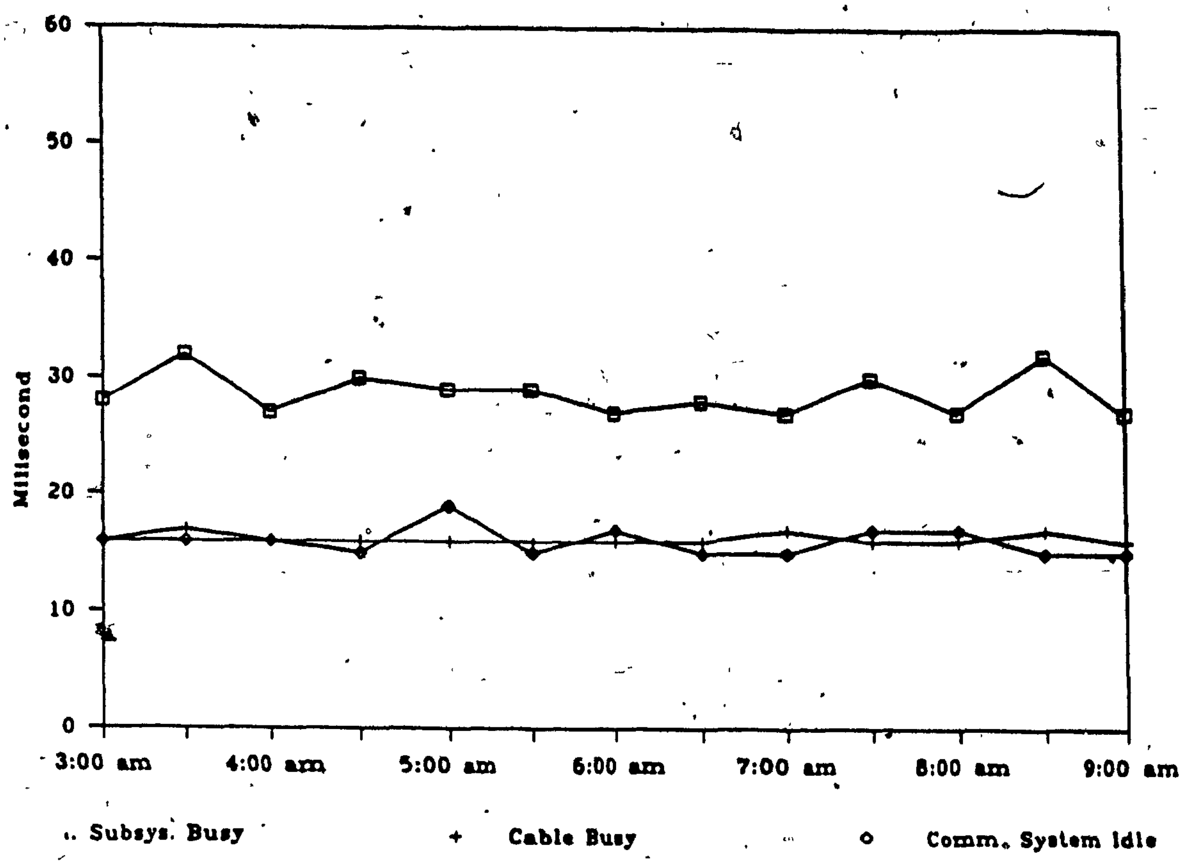


Figure 4.5. Dgram Channel Propagation

Stream / Dgram	Communication System Idle	Cable Busy	Communication Subsystem Busy
Real Time	99.0 / / 16.0	99.0 / / 16.2	99.8 / / 28.7
System Time	4.7 / / 11.5	13.0 / / 11.7	11.4 / / 11.5
User Time	0.9 / / 1.1	0.7 / / 1.1	0.8 / / 1.2

Figure 4.6. Stream / Dgram Channel Performance

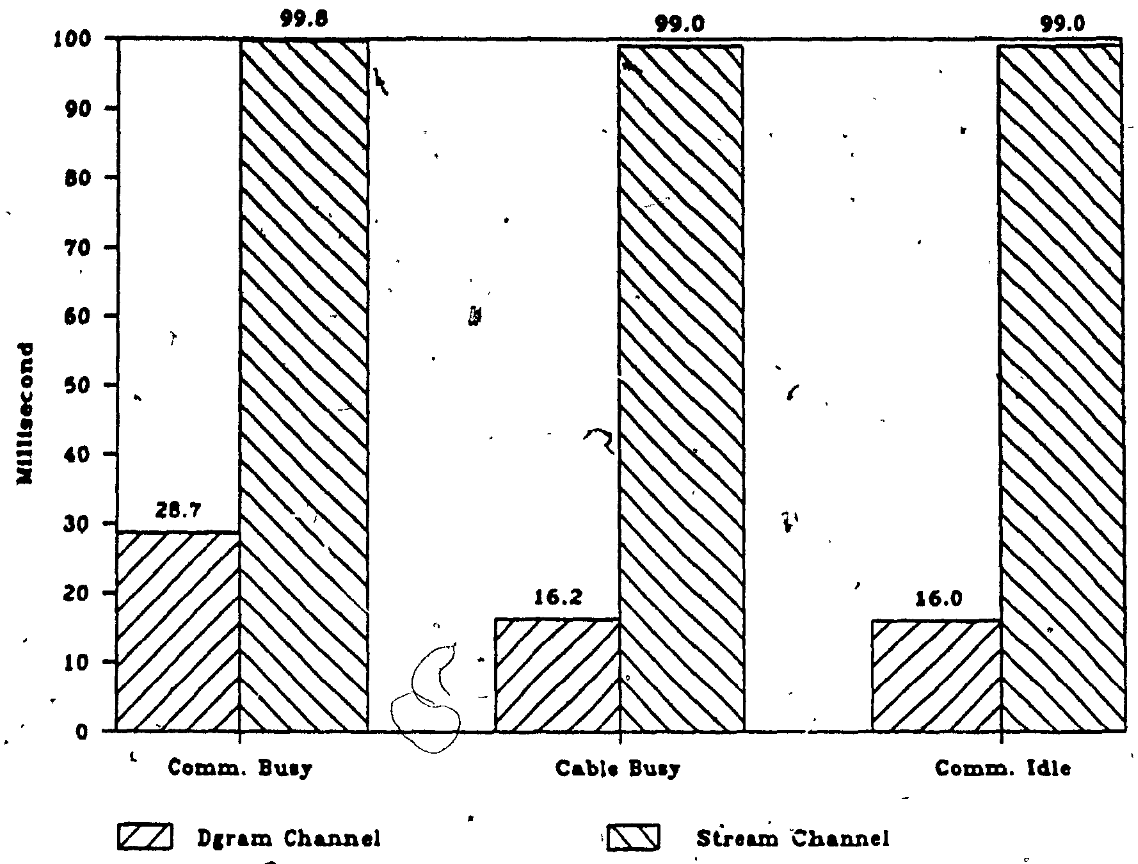


Figure 4.7. Communication Performance Comparison

From these measurements, we observe:

- Message propagation time for Dgram channels is Six Time faster than that of Stream channels;
- The physical cable is not the Bottleneck of the communication system;
- Because of the NFS server, it is difficult to saturate the communication subsystem at each work-station.

The local area network has high bandwidth, low propagation time, and extremely low error rates. However, it is impossible to achieve totally reliable communication. From our measurement, we see the price we have to pay in order to achieve "perfect" communication. Message transmission time is six time slower. And even worse, according to our observation, it still does not achieve 100% delivery. This result reveals that fewer support requirements should be placed on the lower communication layer, and the application layer should have more intelligence to handle communication errors to improve the overall performance.

4.2.3. Performance Characteristics

Our measurements are based on diffused computations. We assume there are N application processes and E communication channels in a diffusion computation. We analyze the speed of a detection algorithm by deriving the time interval between the initiation of a detection and the final compilation of the consistent global state. We define the message complexity as the number of marker message transmitted in the communication system. Obviously, the twp complexities are inter-related.

As pointed by [Venkatesh88], the worst case message complexity for CLA and VLRc-non are $O(E)=O(N^2)$[4], and for VLRc is O(N). It is not difficult to discover that the time

[4]To send a marker onto every channel is not necessary. See chapter 3.2.2.

complexity for CLA is $O(E)=O(N^2)$, for VLRc and VLRc-non are $O(N)$. Figure 4.8 summarizes the time and message complexities of these three algorithms. However, these abstract complexity do not reveal the actual performance, as the proportional constant in the complexity measures do vary drastically between two algorithms or between two different environments.

	Time Complexity	Message Complexity
CLA	$O(E) = O(N^2)$	$O(E) = O(N^2)$
VLRc	$O(N)$	$O(N)$
VLRc_non	$O(N)$	$O(E) = O(N^2)$

Figure 4.8. Algorithm Complexities

We apply CLA, CLA_non, VLRc, and VLRc_non to detect termination of a diffused computation. CLA_non is Chandy's CLA algorithm but using lossy & non-FIFO communication medium. It is only used for performance comparison. CLA does not support lossy & non-FIFO communication environment. However, we found in the course of our test that messages in Dgram channel were seldom lost or delivered out of sequence. For the purpose of comparison, we accept the measured data of CLA_non only in case every message has been correctly transferred.

Process Topology:

Process topology affects the marker traffic, which in turn affects the speed of the detection algorithms. Figure 4.9 and Figure 4.10 are two process topologies used in our performance measurement.

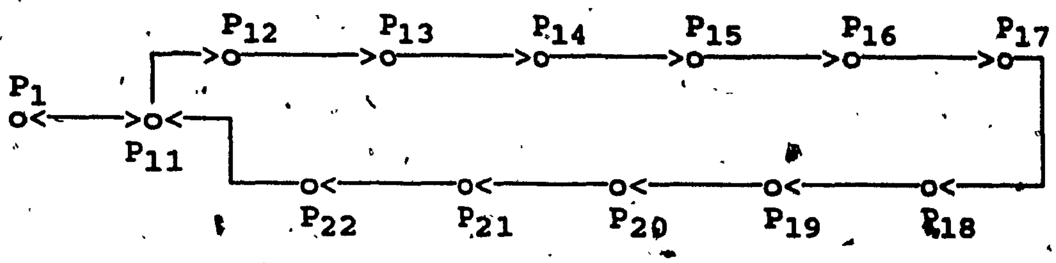


Figure 4.9. Application 1

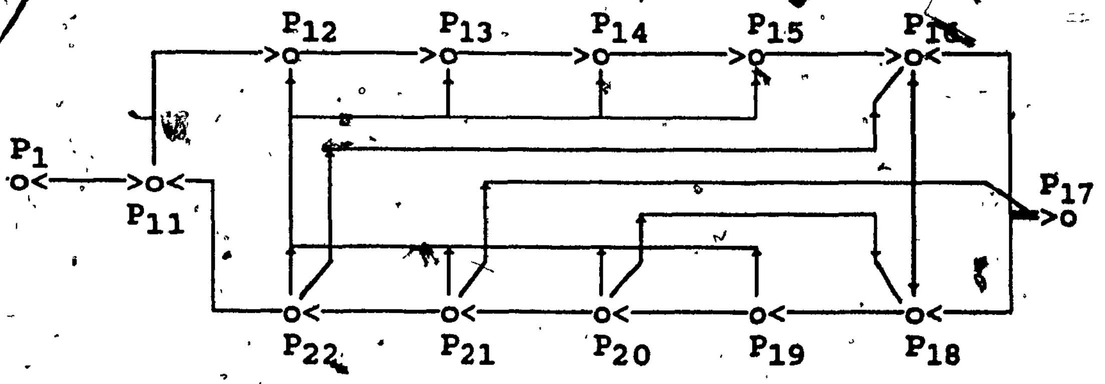


Figure 4.10. Application 2

In application 1 and 2, 13 application processes are distributed among 5 workstations. Process P_1 is designated to be coordinator running at workstation 1; Process P_{11}, P_{15}

and P_{19} are running at workstation 2; Process P_{12}, P_{16} and P_{20} are running at workstation 3; Process P_{13}, P_{17} and P_{21} are running at workstation 4, and process P_{14}, P_{18} and P_{22} are running at workstation 5. Altogether 14 channels exist in application 1, and 37 channels in application 2.

We use application 1 to check, in a real computation, how much time we can gain under the non-FIFO channel environments. From the performance point of view, the difference between CLA and VLRc is that in CLA, the PM will send its local information to the coordinator after it has received one coordination message from each input channel. We call the time interval from the recording of the local process state by a PM to the sending of the local information to the coordinator the **Local Information Detention Time.** Application 2 is designed to maximize the local information detention time.

Test Parameters:

The speed of the detection algorithms depends on the performance of the communication system. The performance of the communication system depends on message traffic. We apply the following three different Test Cases to control the message traffic in the system:

(a) After receiving a message, a diffused process sends one message to each of its output channels, and then it blocks itself. The coordinator P_1 initiates a state detection infrequently (every 5 seconds);

(b) After receiving a message, the diffusion process sends one message to each of its output channels and blocks itself. The coordinator P_1 initiates a state detection often (every 2 seconds);

(c) After receiving a message, the diffusion process sends five messages to each of its output channels and then blocks itself. The coordinator P_1 initiates a state detection often (every 2 seconds).

Figure 4.11 and Figure 4.12 contain the measured data for applications 1 and 2 respectively. The time unit used in both figures is milliseconds. The **State Detection Time** is real time interval in the coordinator station between the invocation of collect_state() and the derivation of the consistent global state. The **Local Information Detention Time** is an average real time interval measured in each process between the invocation of state_recorder() and the sending of the local information to the coordinator. The **Application Message** is the total number of application messages be sent out during the diffusion computation. The **Total Marker** is the total number of marker messages sent out during the diffusion computation. The underling SUN network is too fragile to undertake the test course (c) when using a non-FIFO message channel. All of the measured data are from the average of 25 tests.

		State Detection Time	Local Information Detention	Application Message Sent Out	Total Marker Sent Out
CLA	a	1134	22	196	276
	b	1046	27	196	828
VLRc	a	828	3	196	270
	b	1087	4	196	870
CLA_non	a	461	22	196	276
	b	508	29	196	828
VLRc_non	a	483	6	196	276
	b	512	5	196	828

Figure 4.11. Performance under Case 1

		State Detection Time	Local Information Detention	Application Message Sent Out	Total Marker Sent Out
CLA	a	1135	202	541	110
	b	1322	267	541	660
	c	3157	573	5185	1980
VLRc	a	992	4	541	85
	b	874	4	541	510
	c	2586	9	4927	1615
CLA_non	a	841	246	541	110
	b	827	220	541	660
	c	2777	599	5185	1980
VLRc_non	a	612	126	541	110
	b	740	66	541	660
	c	2137	57	5185	1980

Figure 4.12. Performance under Case 2

Analysis of the Results

We intend to examine the relationship among the communication environment, the message traffic, and the speed of the detection algorithms. The influence of the communication environment and the message traffic on the speed are tested. The speed is measured at the coordinator P_1 which is the only user process running on workstation 1 during the testing. The measured time interval at P_1 is equal to the time consumption for the detection algorithm, plus the time spent waiting for the local information. This time can be obtained by calling the UNIX function Getusage(). However, the time spent waiting for the local information cannot be obtained directly. In order to include this time, we have to use real clocks in our performance testing.

Because of the use of real clocks, it is inevitable that the result also includes some internal expenses which is not caused by the detection algorithm, such as process swapping, resource scheduling, and some system management functions. These uncontrollable factors are the cause of high variation in our measured data, even if we use the average value of the test results.

The conclusions from our measured data are:

(a) The detection algorithms that use lossy and non-FIFO message channels are faster than those that use reliable and FIFO message channels. This suggests that allowing the application layer handle communication errors will achieve better performance;

(b) VLR performs better than CLA on fairly conventional distributed systems, where communication is expensive compared to local communication and large messages are transmitted more efficiently than small ones. From our observation, the time needed by CLA is almost equal to that needed by VLRc plus the local detention time. In other words, it is the local detention time that makes CLA slower than VLRc;

(c) When the communication system is saturated, VLRc algorithm is better than CLA algorithm. This can be explained as the message complexity of CLA is O(E), while, the message complexity of VLRc is O(N). When measuring the test case (c) of application 2, the local detention time is double of that of test cases (a) and (b). The increase of message propagation time causes an extra delay in global state detection.

Chapter V. Distributed Garbage Collection

High performance for many applications is achievable in parallel or distributed systems. Such applications are often written in applicative or logic programming languages, both of which require garbage collection. In some imperative languages, such as Lisp, SmallTalk, and CLU, memory space is not deallocated explicitly and so garbage collection is required. Therefore, garbage collection in a distributed computation is an important problem [Hudak82] [Walker84] [Hughes85].

In this chapter, we will present an efficient garbage collection algorithm for collecting obsolete objects in the language CLU in a DCS environment. This algorithm is a derivative of our Probabilistic global state detection algorithm —— each Node (computer) of the system is responsible for its own collection.

5.1. CLU Language Introduction

CLU [Liskov77] [Liskov84] is an Object-Oriented programming language, which regards programs as operating on potentially ever-lasting objects. Usually, programs are developed by means of problem decomposition in a top down fashion. CLU provides programmers with abstraction mechanisms for program design and implementation. Compared with conventional languages which support only **Procedure Abstraction**, CLU provides, in addition to procedures, linguistic mechanisms that support the use of **Data Abstraction** and **Control Abstraction**.

Data Abstraction is used to specify a new type of data object. It may consist of a set of objects and a set of operations that characterize the behaviour of the objects, such as object creation, object read-out and object modification. **Control Abstraction** is used to sequence the needed actions, similar to the operation of **If**, **While**, and **For** statements in all other languages. In addition, CLU allows the user to define repetition methods. A **Procedure Abstraction** specifies a computation on a set of input objects, and the production of a set of output objects.

Program Structure

CLU supports structured programming in which problems are decomposed based on abstractions (objects and operations). Each abstraction may be further decomposed. An abstraction isolates the use from the implementation, i.e., an abstraction can be used without knowledge of its implementation, or implemented without knowledge of its use.

A CLU program consists of a group of **Modules**. Each module is either a single abstraction or, if **Parameterized**, a class of related abstractions. Modules are never embedded in each other. Modules are a separate textual unit, and can be compiled independently of one other. There are three types of modules in CLU:

Procedure	———	Supports procedure abstraction;
Iterator	———	Supports control abstraction;
Cluster	———	Supports data abstraction.

A Procedure performs an action on objects, and terminates by returning objects. Arguments are Shared between the caller and the invoker, known as Call By Sharing. A procedure has no global variable unless it is defined inside a cluster. An Iterator is similar to a procedure. It is used to compute a sequence of objects based on its arguments. Objects in the sequence are provided to the caller one at a time. A Cluster implements a data abstraction. It consists of a set of Objects and a set of Primitive Operations to create and to manipulate these objects. Operations can be either procedures or iterators. Procedures, iterators, and clusters can all be parameterized. Parameterization provides the ability to define a class of related abstractions in a single module, which implies that the module was written without knowledge of the actual types of parameters.

Program Semantics

The basic elements of CLU semantics are **Objects** and **Variables.** Objects are data entities that are created and manipulated by programs. Variables are object names (Pointers) used in a program to refer to objects.

Each object has a Type, which characterizes its behaviour. A type defines a set of primitive operations to create and to manipulate objects of that type. An object is created and manipulated only via the operations of its type. Objects can have components which are other objects. The existence of the objects is Independent of the activities of the procedures and iterators. The space for the objects is allocated from a dynamic storage area (Heap), and it is Not Deallocated explicitly. In practice, the heap space occupied by an object can be reclaimed after the object is no longer Accessible by a CLU program.

CLU variables are object names used in a program to Denote objects at execution time. Unlike variables in many other languages, where variables are containers of values, CLU variables are object names rather than the objects themselves. As a result, it is possible for two variables to denote the same object.

There are two basic actions in the CLU language: Assignment and Invocation. An assignment primitive X:=E causes variable X to denote the object resulting from the evaluation of expression E. Assignment neither copies the object nor affects the state of any object. The invocation involves assigning arguments through assignment. Objects are shared between the caller and the invoker.

Distributed CLU

We now suggest a CLU implementation under the DCS environment, called Distributed CLU. Suppose we have a distributed computer system with N_0 loosely synchronized nodes (computers). Nodes are connected by message channels (no shared memory). In this system, the heap is distributed and has partitions residing at all N nodes. We assume that memory space for CLU objects is allocated from this Distributed Heap, and objects can be referred to uniformly regardless of their location. Obviously, there should be a mechanism for accessing objects by their Unique object names. We assume that after an object is created at a node, it will not move to other nodes. However, object names are permitted to travel around the whole distributed system.

5.2. Garbage Collection in Distributed CLU

Programming languages such as CLU, Lisp and SmallTalk use a model of computation in which objects reside in a Heap and objects are not deallocated Explicitly. Instead, garbage collection is performed at some convenient time to deallocate the Inaccessible Objects. An implication of such an implementation is that the heap is distributed to individual **Nodes** (computers), and that objects are accessible at any node regardless of their location. In this section, we will develop a **Garbage Collection Algorithm** for memory resources occupied by inaccessible objects. First of all, we define some terms which will be used in this chapter.

Terminology:

Object: An object is a data entity, together with a set of operations to create and to manipulate itself or other objects. The memory space needed by an object is allocated from a Distributed Heap.

Object Name: An object name is a **Pointer** that a programmer uses to refer to the object. An object name is denoted by a CLU variable, which may migrate to other nodes.

Local Object: An object is local to node N_i ($1 \le i \le N$), if it is inaccessible at other nodes, or if its space is allocated from a heap partition that belongs to N_i.

Remote Object: An object is remote to node N_i ($1 \le i \le N$), if it is accessible at N_i, and the object name was either received from another node, or sent to other nodes.

Public Object: An object is public to node N_i ($1 \le i \le N$), if its accessibility at other nodes is uncertain, and the object space is allocated from the heap partition that belongs to N_i.

Local Garbage Collection: Local garbage collection L_i ($1 \le i \le N$) is a garbage collection process at node N_i. L_i reclaims the heap space of the inaccessible local objects of N_i.

Public Garbage Collection: Public garbage collection is a garbage collection process. When a node wishes to eliminate its public object (O), it asks this process to find out from a most recently detected global state to see if O is accessible at any other nodes.

5.2.1. Distributed Garbage

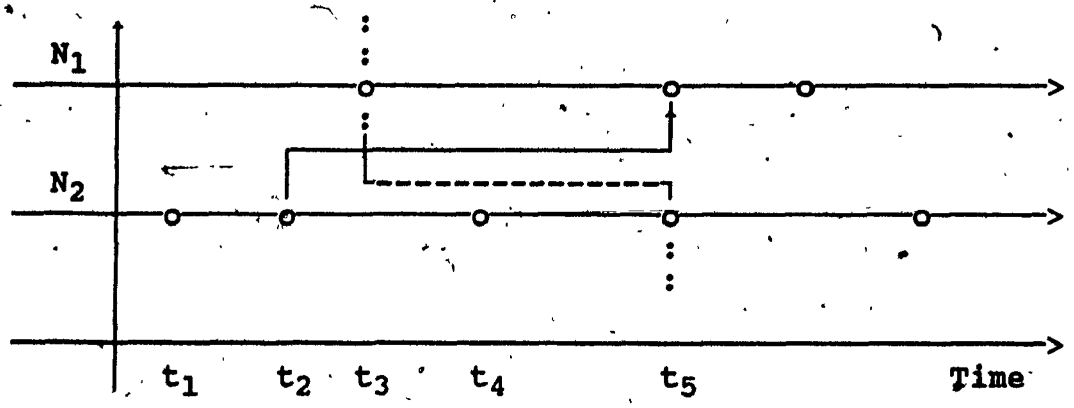
The garbage in this context is the memory space allocated to the Inaccessible Objects. In non-distributed systems, an object is accessible if it is denoted by variables of an active procedure, or if it is a component of an accessible object.

In a DCS environment, however, the object accessibility is not that intuitive. First, objects can be referred to at any node. An inaccessible object at one node may be accessible at another node. Secondly, object names may be in transit. Because of the delay of message transmission, an inaccessible object at one instant may become accessible at some other instant. This special problem in distributed systems is known as the Transit Problem, which we have seen already in the context of a channel state in the global state.

As shown in Figure 5.1, at t_1, node N_2 is the only node at which object X may be referred to. Suppose after sending the object name of X to node N_1, N_2 deletes its own reference to X and N_1 receives the object name of X at t_5. If N_1 performs local garbage collection at t_3, it will find X was inaccessible at N_1; and if N_2 performs local garbage collection at t_5, it will find object X was inaccessible at N_2 too. If the global garbage collection has gone through the above, it may decide object X is inaccessible at both N_1 and N_2, and destroy object X incorrectly. Indeed, even if an object is inaccessible at any node, it may not be a garbage if a reference to the object is in transit. To handle the transit problem in distributed systems, we define accessibility as follows:

Definition 5.1:

In the DCS environment, an object is accessible if it is either accessible at one of N nodes, or its name is in transit. If an object name is in transit in a channel, we say that the object is accessible from that channel. ∎



t_1: Object X is accessible at N_2 only
t_2: N_2 sends the name of X to N_1
t_3: N_1 performs local garbage collection
t_4: Object X becomes inaccessible at N_2
t_5: N_1 receives the object name of X
N_2 performs local garbage collection

Figure 5.1. Transit Problem

5.2.2. Garbage Collection vs Global State

The global state detection and the distributed garbage collection problems are related. Intuitively, the distributed garbage collection needs global information about the whole system. If there is a global state available, then the accessibility of objects at each node and channel can be revealed. This implies that distributed garbage collection would have the same requirements as global state detection. In particular, local garbage collection at each node is synchronized, and the global garbage collection compiles the information from all nodes [Ali84] [Hughes85] [Liskov86].

In [Li87], global states have been classified according to the usage requirements. In order to determine the kind of global state needed for a distributed garbage collection, we need to examine the inherent properties of object accessibility. The inaccessibility is a **Stable Property** [Liskov86], i.e., an inaccessible object remains inaccessible until it is deallocated. The stable property can be revealed by a stable global state [Li87]. However, our accessibility is defined on both processes and channels and a stable global state does not contain enough channel information. Thus, a stable global state is not suitable for the distributed garbage collection.

The transit problem makes garbage collection in distributed systems much more difficult than in centralized systems. There are different approaches to solve the transit problem. Some distributed garbage collection algorithms [Ali84] [Hudak82] are only valid for tightly coupled parallel architectures where the message transmission is instantaneous. Then a stable global state is sufficient for detecting the accessibility of objects. Some other algorithms make impractical assumptions, for examples, the existence of a common clock in the distributed systems [Hudges85], or bounded delay time for message passing [Liskov86].

In order to solve the transit problem, we will apply a **Two-Phase-Commit** Protocol in our distributed garbage collection algorithm. Either the sender or the receiver should remember the transaction involving an object name transmission until the transaction has completed.

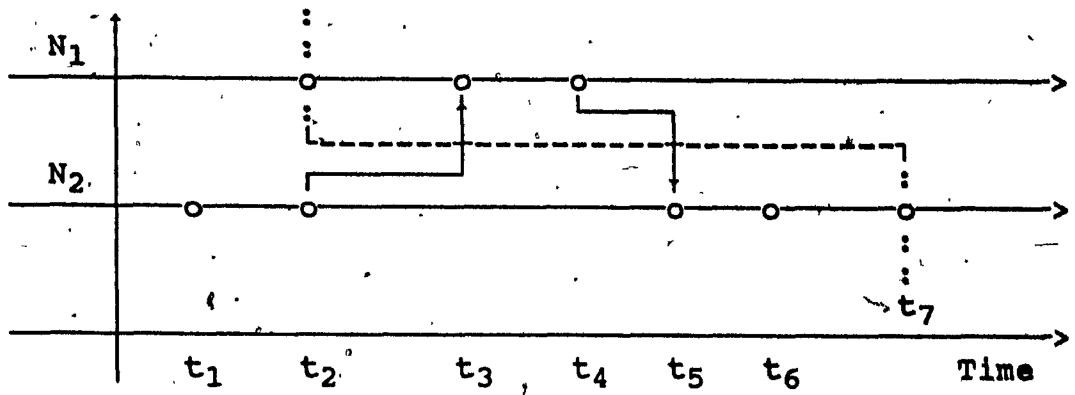
Two-Phase-Commit Protocol for Object Name Transmission:

At the Sender's Site: When sending an object name to another node, the sender marks the object in-transit; after receiving the **Acknowledge Message** from the receiver, the sender removes the mark.

At the Receiver's Site: Upon receiving an object name, the receiver returns an acknowledge message to the sender.■

Unfortunately, the two-phase-commit protocol does not solve the transit problem completely. If the information about the object accessibility at each node is collected independently, then there may exist the case of incorrect garbage collection. For example, in Figure 5.2, N_2 is the only node that may refer to object X at t_1. Suppose N_2 sends the object name of X to node N_1 at t_2, and then receives the acknowledge message from N_1 at t_5. N_2 releases its reference to object X at t_6. If a global garbage collection makes use of the information from N_1 at t_2 and N_2 at t_7, then object X will be incorrectly destroyed. This is because at t_2, object

X is not accessible at node N_1 and not marked in-transit, and at t_7, object X is not accessible at node N_2 and is not marked in-transit.



t_1: Object X is accessible at N_2 only
t_2: N_1 performs local garbage collection
N_2 sends the name of X to N_1
t_3: N_1 receives the object name of X
t_4: N_1 send the acknowledge message to N_2
t_5: N_2 receives the acknowledge message
t_6: Object X becomes inaccessible at N_2
t_7: N_2 performs local garbage collection

Figure 5.2. Two-phase-Commit

If we examine the transit problem carefully, we will discover that the cause of the above error is that of inconsistency between the channel states and the process states. In this inconsistent global state, an effect event (N_2 having received an acknowledge message) is included in the global state, but not the cause event (N_1 sending the acknowledge to N_2). However, if the global garbage collection has used a consistent global state, then either the object X will be marked accessible at a node, or its name will be marked as in transit.

From the above discussion we can conclude that when using the two-phase-commit protocol, the transit problem can be solved if and only if the global state for the global garbage collection is a consistent global state. Here, a node state is a list of accessible remote objects and a list of public objects whose accessibility is uncertain. A channel state is a list of objects whose name has been sent out but for the sender has not received the acknowledge message from the receiver.

5.2.3. Probabilistic Collection Algorithms

We now present our garbage collection algorithms for distributed CLU objects. Our method allows each node to perform local garbage collection Independently of other nodes without communicating with other nodes which contain the other parts of the heap. In particular, different nodes can use different garbage collection techniques, and do local garbage collection at any time.

In our algorithm, each node will maintain a list of its **Public Object** names and a list of its **Remote Object** names. The garbage collection at each node is responsible for reclaiming its inaccessible local objects, and deleting its inaccessible remote object names. At the end of a local garbage collection, the node will inform other nodes of its accessibility of the remote objects. The accessibility of public objects is detected by the global garbage collection.

Every node needs global information for global garbage collection. In order to have better performance under the DCS environment, we adopt the working model proposed by [Liskov86]. In this model, nodes do not communicate with one another. Instead, information about inter-node references is stored in a Logical Central Service (LCS). The node communicate with the LCS periodically to update the accessibility of the remote objects and to examine the accessibility of its public objects.

Nodes provide the LCS with information about their remote object accessibility and public object names. The LCS determines accessability of these public objects. An innovative part of our algorithm is that each node communicates with the LCS independently. Local garbage collections are performed asynchronously.

Data Structure:

each node N_i ($1 \le i \le N$) maintains two lists as **Node State:**

Access-list: <remote-object-name>

When a remote object is accessible from N_i, its name is in Access-list (i.e., when an object name is received, its name is added to this list, and after local garbage collection, the inaccessible remote objects at N_i are deleted from this list);

Public-list: <public-object-name, global-time>

Every public object has an entry in this list (i.e., when a local object becomes a public object, its object name and the global time of creation are added to this list, and after global garbage collection, the inaccessible public objects with the same global clock label are deleted from this list); ■

Each node N_i ($1 \le i \le N$) maintains a list as **Channel State:**

Transit-list: <object-name, target-node>

After an object name has been sent out and before the acknowledge is received, the object name is in the list. ■

There are two **Data Structures** in our algorithm:

Local-state:<node-state,channel-state,global-time,node-id>

The node node-id at indicated global-time recorded this node-state and channel-state;

Garbage-list: <public-object-name, global-time>

The LCS informs a node that a public object created at the indicated global-time is garbage. ■

Algorithm:

The accessibility of the public objects is detected at the LCS by analyzing a consistent global state. We use the Probabilistic Algorithm for the consistent global state detection. The probabilistic detection algorithm has already been discussed in chapter three.

We assume that there is a global time system underlying the user computation. Actions which are regarded as events in the Global Time System are: sending and receiving an acknowledge message, and the process state recording. Our distributed garbage collection algorithm deals with the detection of public object accessibility.

Algorithm 5.1.

Local Garbage Collection at Each Node N_i $(1 \le i \le N)$:

```
Begin
   do local object garbage collection;
   for every object-name ∈ Access-list
      if it is no longer accessible from N_i then
         delete it from Access-list;
   record a local state;
   send the local state to LCS;
   while there is a G := Garbage-list from LCS
      for every <object-name, global-time> ∈ Public-list
         if <object-name, global-time>
                 = G<object-name, global-time> then
            delete it from Public-list;
            declare it as garbage;
End;
```

■

Algorithm 5.2.:

Global Garbage Collection at LCS:

```
Begin
 do forever
   read a local state LS from any node;
   store the LS into the local state database DB;
   if find a new consistent global state GS in DB then
      for every $N_i$ do:
       Begin
         let garbage-list G := empty;
         for every <object-name, global-time> $\epsilon$
                GS< <Public-list>, $N_i$ > do:
            if the object is inaccessible from any node
                       and channel in the GS then
               add the object to garbage-list G;
         Send the Garbage-list G to node $N_i$;
         for every local state $LS_j$ of $N_i$ $\epsilon$ DB
            if global time of $LS_j$ « global time of $GS_i$
               delete $LS_j$ from DB;
       End;
End;
```

The crucial part of our algorithm is to detect a consistent global state from a set of local states which were recorded synchronously at different nodes. From **Theorem 3.4**, if none of events leading to the recording of a local state depends on others yet to be accounted for, in the record states, then the recorded global state is consistent. So the following algorithm will detect a consistent global state from the local state database DB.

Let LS_{ij} be the jth global state of node N_i $(1 \le i \le N)$. Let LS_{i0} be the oldest local state of node N_i in DB. Let LS_{iMAX} be the latest local state of node N_i in DB.

Algorithm 5.3.:

Consistent Global State Collection from DB:

```
Begin
   GS = empty;
   for each i ($0 \le i \le N$) do:
      GS := GS U ( $LS_{iMAX}$ );
   for each i ($0 \le i \le N$) do:
      Begin
        GS' := GS;
        $LS_{ij}$ := local state belongs to GS;
        if ( !find_GS( $LS_{ij}$) ) then
             return( false);
        if ( GS != GS' ) then
             i := 0;
      End;
   return( true);
End;


find_GS( $LS_{ij}$)

   Begin
      for each l ($0 \le l \le N$) do:
        Begin
          $LS_{lm}$ := local state belongs to GS;
          for each k ($N \ge k \ge 0$) do:
             if ( GC($LS_{lm}$) « GC($LS_{ij}$) ) then
                  k := -1;
          if ( k != -1 )
                  return( false);
          GS := GS - ( $LS_{lm}$ );
          GS := GS U ( $LS_{lk}$ );
        End;
      return( true);
   End;
```
■

The time complexity of the above consistent global state collection algorithms is $O(\sum_i \sum_j LS_{ij})$ ($1 \le i \le N$, $0 \le j \le MAX$), since any local state in DB could be in GS only once. Function find_GS(LS_{ij}) returns a global state in which none of the global clock labels of the local states are greater than ("«") the GC(LS_{ij}).

5.2.4. Simulation Report

Our probabilistic algorithm has no overhead for coordination and synchronization, but it requires additional local state recordings. The number of additional local state recordings is the most important performance characteristic of our probabilistic algorithm. In order to study such a factor, we implemented algorithm 5.3 and simulated algorithm 5.2 on a SUN workstation.

The probability of finding a consistent global state from a set of local states depends on several things. Among them, the process topology and the probability of occurrence of local state recording events are crucial and are simulated.

Process Topology: Obviously, the more message channels and processes are there in a computation, the higher the probability will be that two local state recording events are dependent, therefore, the more additional local states will be needed in order to find a consistent global state. For the channel connection, we simulated the worst case scenario —— each process can send messages directed to other processes (complete connection) and a better case scenario —— each process can only directly send messages to one process (linear connection). In our simulation, the number of processes is varied from five to thirty.

Local State Recording: In our simulation, we assume that the distribution of message sending events and local state recording events are even. The probability of occurrence of a local state recording event between two message sending events is varied from 1/4 to 1/32.

Figure 5.3 shows the simulation results for the complete connection and Figure 5.4 shows the simulation results for the linear connection. Here, a window means the average number of local states for finding a consistent global state.

		Number of Processes					
		5	10	15	20	25	30
Probability of Local State Recording	1/4	2.1	1.9	1.8	1.8	1.7	1.6
	1/8	3.8	3.6	3.0	2.8	2.4	2.1
	1/12	6.7	8.2	6.0	4.8	3.4	3.4
	1/16	10.8	20.1	11.5	9.3	6.1	4.1
	1/20	23.7	32.5	38.2	17.7	9.1	7.8
	1/24	43.0	57.0	44.5	31.0	18.0	11.2
	1/28	51.0	57.0	60.0	47.0	45.0	33.2
	1/32	54.0	57.0	60.0	60.0	51.0	46.0

Figure 5.3. Window Size in Complete Connection

Probability of Local State Recording	Number of Processes					
	5	10	15	20	25	30
1/4	2.2	2.0	1.9	1.9	1.7	1.7
1/8	3.4	3.5	3.3	2.9	2.5	2.3
1/12	5.7	6.8	6.2	4.7	4.1	3.2
1/16	11.8	13.0	12.3	8.8	5.6	4.4
1/20	18.0	26.0	31.2	15.5	10.1	8.7
1/24	26.0	44.0	40.0	34.0	19.9	13.1
1/28	37.0	50.0	54.0	54.0	36.0	27.5
1/32	55.5	57.0	60.0	54.0	44.0	40.0

Figure 5.4. Window Size in Linear Connection

From our simulation results we can find that the probability of occurrence of a local state recording event between two message sending events heavily affects the window size. This suggests to us that if local states can be recorded within a small time interval, then the window size will be relatively small.

5.3. Discussion

The distributed garbage collection algorithm discussed above allows the heap space for distributed CLU objects to be reclaimed in an efficient manner. This algorithm can actually be used in the general problem —— distributed dynamic resource allocation, such as orphan detection [Walker84], task deletion [Hudak82], and inconsistent replicated data detection. Our algorithm has several performance advantages. First, the garbage collection at each node is totally Independent. Because local garbage collection at each node is asynchronous, the node can perform local garbage collection at any time, for example, as a background task.

Second, our algorithm separates local and global garbage collection. The latter is performed in the LCS. We all know that the amount of garbage collection can be substantial. So this problem decomposition may lead to better overall computation efficiency, and may prove especially valuable in a real-time system. In general, our algorithm does not cause delays ~~of~~ user computations except when recording local state, which is not time costly.

Our algorithm is a probabilistic algorithm in the sense that global garbage may not be collected unless the LCS finds a consistent global state. To improve the probability of success, the following suggestions may be considered:

- If the communication system supports message broadcasting, a node sends a signal to ~~all~~ other nodes before recording its local state. Upon receiving such a signal, each node should record its local state as soon as possible.

- Make local state recording in a **Burst Mode**, i.e., when a node records its local state, it will record its local state several times within a short time period.
- In the applications where garbage should have a maximum lifetime, the probabilistic algorithms can be used in conjunction with coordinated algorithms to meet the requirement while retaining a reduced synchronization overhead.

Chapter VI. Conclusions

6.1. Consistent Global State

The concept of consistent global state in distributed computer systems was formalized in [Chandy85] and in [Li87]. The most important property of a consistent global state is that of Causality Preservation —— if an event is included in a global state, then its cause event is also included in the global state. Consistent global states possess some other properties such as Reachability, Recoverability, and Global Invariant Preservability. Efficient solution to problems like rollback recovery, dynamic resource allocation, fault-tolerant computing, and distributed debugging can be derived using these fundamental concepts.

The problem of consistent global state detection can be regarded as an extension of the generic problem of distributed clock synchronization. A simple solution to this problem is to assume a common clock in distributed systems. However, this assumption is impracticable. In this thesis, we presented four different time systems, namely, **Theoretical Time System, Synchronized Real Time System, Logical Time System,** and **Global Time System**. We proved that any **Instantaneous Global State** in terms of the above time systems is a consistent global state. More importantly, the instantaneous global state in terms of the above time systems (except the theoretical time system) can be captured practically.

According to the **Degree** of process cooperation, we classified consistent global state detection algorithms into three categories, namely **Punctual Algorithms, Coordinated Algorithms,** and **Probabilistic Algorithms**. Punctual algorithms are a conceptual extension of the time-out interruption of uni-processor systems. They are easily applied. Punctual algorithms require synchronized local clocks on different computers. Most of consistent global state detection algorithms examined were coordinated algorithms. The coordinated algorithms involve coordination of the other processes to perform global state collection. Obviously, a coordinated detection algorithm has synchronization overhead. The synchronization overhead is expensive in distributed computer systems. A probabilistic algorithm allows the processes perform their local state recording asynchronously and so does not incur any synchronization overhead. However, a probabilistic algorithm does not guarantee the detection of a consistent global state and may require more redundant local state recording.

The performance study of the coordinated detection algorithms presented in chapter four is experimental. The measured data indicated that inter-computer communication is expensive, especially when perfect transmission is needed, or when the communication system is saturated. Our performance study revealed that distributed algorithms can achieve better performance if the application layer can handle some communication errors. According to our observation, the rate of message transmission is very low. We also conclude that the VLR algorithm has better performance characteristics than the CLA algorithm.

Garbage collection in distributed computer systems is an important problem. It requires gathering global information. If resources or their pointers are allowed to migrate from a node to another, the global state for resource deallocation must be Consistent. A probabilistic detection algorithm is a better candidate than other algorithms, because of the reason already mentioned.

6.2. Contributions and Related Work

This thesis studied the consistent global state of a DCS extensively. We viewed a consistent global state as a sort of instantaneous global state in terms of a time system. We presented four time systems for the purpose of consistent global state detection. Moreover we classified detection algorithms into three categories, and applied the probabilistic detection algorithm to distributed garbage collection. Based on the discussion in [Li87] and [Venkatesh88], we implemented an experimental global state kernel for a performance study.

The problem of distributed clock synchronization is viewed as an extension of the generic problem of consistent global state detection. Global Time was formulated after examining the relationship between the consistent global state and the physical theory of time. Ideas similar to global time have been used in a number of places: in [Walker84], it was used to detect crashed nodes; in [Liskov86], it was used to update inconsistent replica of data; in [Passier88] and [Radha87], it was used to support rollback and recovery. The global time system is used to describe the independence of distributed events. We proved the isomorphism between the global time model and the space-time model of distributed computations.

The original idea of a probabilistic algorithm for consistent global state detection belongs to my supervisor Dr. Li. What I did was to design the probabilistic detection algorithm and to apply it to the distributed garbage collection problem. We noticed that there is no literature on the probabilistic detection algorithms.

6.3. Future Work

We have presented a probabilistic algorithm for consistent global state detection. This algorithm hopefully can perform better in a DCS environment. Some analytic and experimental evaluation of this approach will be needed to establish the claim.

References

[Alford85] M.W. Alford, J.P. Ansart, et al, Distributed Systems --- Methods and Tools for Specification, an Advanced Course, Lecture Notes in Computer Science, No. 190, ed. M Paul and H. J. Siegert (Berlin Heidelberg New York: Springer-Verlag, 1985), pp. 454-468.

[Ali84] K.A.M. Ali, "Object-Oriented Storage Management and Garbage Collection in Distributed Processing System," Ph.D Thesis, Royal Institute of Technology, Stockholm, December, 1984.

[Chandy83] K.M. Chandy, J. Misra and L.M. Haos, "Distributed deadlock Detection," ACM Transactions on Computer Systems, Vol. 1, No. 2, May 1983, pp. 144-156.

[Chandy85] K.M. Chandy and L.Lamport, "Distributed Snapshots: Determination of Global States of Distributed Systems," ACM Transaction on Computer Systems, Vol. 3, No. 1, February 1985, pp. 63-75.

[Davies81] D.W. Davies, E. Holler, et al, Distributed Systems --- Architecture and Implementation, an Advanced Course, Lecture Notes in Computer Science, No. 105, ed. B.W. Lampson, M.Paul and H.J. Siegert (Berlin Heidelberg New York: Springer-Verlag, 1981), pp. 6-9.

[Davies81a] D.W. Davies, E. Holler, et al, p. 267.

[Dijkstra80] E.W. Dijkstra and C.S. Schelten, "Termination Detection for Diffusing Computation," Information Processing Letters, Vol. 11, No. 1, August 1980, pp. 1-4.

[Elmargar86] A.K. Elmargarmid, "A Survey of Distributed Deadlock Detection Algorithms," SIGMOD RECORD, Vol. 15, No. 3, September 1986, pp. 37-45.

[Fischer82] M.J. Fischer, N.D. Griffeth, and N.A. Lynch, "Global States of a Distributed System," IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, pp. 198-202.

[Gligor80] V.D. Gligor and S.H. Shattuck, "On Deadlock Detection in Distributed System," IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, September 1980, pp. 435-440.

[Gusetta87] R. Gusetta and S. Zatti, "TSP: The Time Synchronization Protocol for UNIX 4.3BSD," System Manages Manual for 4.3BSD, University of California, Berkeley, Section 22.

[Hudak82] P. Hudak and R.M. Keller, "Garbage Collection and Distributed Application Processing System," Proceedings of the ACM Symposium on Lisp and Functional Languages, August, 1982, pp. 168-178.

[Hughes85] J. Hughes, "A Distributed Garbage Collection Algorithm," Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, No. 210, ed. G. Goos and J. Hartmanis (Berlin Heidelberg New York Tokyo: Springer-Verlag, 1985), pp. 256-272.

[Jefferson85] D.R. Jefferson, "Virtual Time," ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, July 1985, pp. 401-425

[Lai87] T.H. Lai and T.H. Yang, "On Distributed Snapshots," Information Processing Letters, Vol. 25, No. 3, May 1987, pp. 153-158.

[Lamport78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, Vol. 21, No. 7, July 1978, pp. 558-565.

[Lamport85] L. Lamport and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," Communications of the ACM, Vol. 32, January 1985, pp. 52-72.

[Li87] H. F. Li, K. Venkatesh and T. Radhakrishnan, "Global States of Distributed Systems," Submitted for publication in IEEE Transactions on Software Engineering.

[Liskov77] B. Liskov, A. Snyder, et al, "Abstraction Mechanism in CLU," Communication of the ACM, Vol. 20, No. 8, August, 1977, pp. 564-576.

[Liskov84] B. Liskov, R. Atkinson, et al, CLU Reference Manual, (Berlin Heidelberg Tokyo: Springer-Verlag, 1984), pp. 1-15.

[Liskov86] B. Liskov and R. Ladin, "Highly Available Distributed Services and Fault Tolerant Distributed Garbage Collection," Proceedings of the Fifth Annual Symposium on Principles of Distributed Computing, August 1986, pp. 29-39.

[McCarthy87] D.D. McCarthy, "Time," McGraw-Hill Encyclopedia of Science & Technology, 1987 ed.

[Morgan85] C. Morgan, "Global and Logical Time in Distributed Algorithms," Information Processing Letters, Vol. 20, May 1985, pp. 89-99.

[Natarajan86] N. Natarajan, "A distributed Scheme for Detecting Communication Deadlocks," IEEE Transactions on Software Engineering, Vol. SE-12, No. 4, April 1986, pp. 531-537.

[Passier88] C. Passier, "Experimental Evaluation of Distributed Rollback and Recovery Algorithms," M. Comp. Sci. thesis, Concordia University, Montrèal, Quèbec, Canada, August 1988.

[Pratt86] V. Pratt, "Modelling Concurrency with Partial Orders," International Journal of Parallel Programming, Vol. 15, No. 1, January 1986, pp. 33-71.

[Radha87] T. Radhakrishnan, H.F. Li and K. Venkatesh, "Concurrent and Domino Free Rollback Recovery in Distributed systems," Proceedings of the IFIP Conference on Distributed Processing, Amsterdam, October 1987.

[Shin87] K.G. Shin and P. Ramanathan, "Clock Synchronization of Logic Multiprocessor System in the Presence of Malicious Faults," IEEE Transactions on Computers, Vol C-36, No. 1, January 1987, pp. 2-12.

[Spezialet86] M. Spezialetti and P. Kearns, "Efficient Distributed Snapshots," Proceeding of the Sixth Conference on Distributed Computing Systems, 1986, pp. 382-388.

[SUN86] UNIX Interface Reference, SUN Microsystems, July 1986.

[Venkatesh88] K. Venkatesh, "Global States of Distributed Systems: Classification and Applications," Ph.D thesis, Concordia University, Montrèal, Quèbec, Canada, January 1988.

[Walker84] E.F. Walker, "Orphan Detection in the Argus System," M.Sci Thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts 02139, U.S.A.