BALANCED SEQUENCING PROTOCOLS

Yeturu Aahlad

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-34 November 1991

BALANCED SEQUENCING PROTOCOLS

APPROVED BY
DISSERTATION COMMITTEE:

James C. Browne

Mohamed Gouda

Hank Korth

Simon Lam

Miroslaw Malek

To my parents,
Yeturu Venkata Satyasena Reddi and Yeturu Saroja Reddi
and my wife, Subha

BALANCED SEQUENCING PROTOCOLS

by

YETURU AAHLAD, B. TECH., M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 1991

Acknowledgements

I am very grateful to my advisor, Dr. J. C. Browne, for his enthusiastic support and encouragement. I thank my dissertation committee members, Drs. Gouda, Korth, Lam and Malek for their guidance. I thank Yvonne Ballester, Nancy Macmahon Karen Nordby and Gloria Ramirez for their help with a bewildering assortment of administrative problems. I thank my family, especially my parents, for their unflinching confidence in my ability to succeed.

BALANCED SEQUENCING PROTOCOLS

Publication No. _____

Yeturu Aahlad, Ph.D.

The University of Texas at Austin, 1991

Supervisor: James C. Browne

The protocol used to control the sequence of execution of events of a distributed computation has a significant impact on its performance. Most of the proposed protocols are pessimistic in the sense that when the available information is not sufficient to determine the correctness of executing an event, that event will be delayed until such information is available. Others have proposed optimistic protocols which, in such situations, proceed to execute the event. When the necessary information becomes available, if it turns out that the event should not have been executed, the protocol takes appropriate action to recover from the mistake.

This research addresses ways to strike an appropriate balance between the extremes of optimism and pessimism in a sequencing protocol and evaluates the benefits of doing so. The term Balanced Sequencing Protocol refers to protocols whose degree of optimism can be varied across a spectrum of possibilities ranging from optimistic to pessimistic by tuning one or more parameters of the protocol. Two approaches are employed in the investigation:

1. a general protocol for sequencing any program at any specified level of optimism and

2. balanced sequencing protocols specialized for some common distributed computing primitives and paradigms, namely, producer-consumer, distributed semaphores and distributed locking.

For these specialized protocols, the range of circumstances where balanced protocols do better than both extremes and the optimal balance are analytically determined. A model of distributed databases used in a previously published simulation experiment is studied, and balanced locking is demonstrated to perform better than conventional locking when recovery cost is less than 20 message delays.

During the course of this research, a previously unknown phenomenon which can cause the performance of optimistic protocols to degrade over time was identified, and its effects were quantified for a simple system. A solution to this problem based on balanced sequencing is proposed.

Table of Contents

Balanced Sequencing Protocols

Yeturu Aahlad

Chapter 1
Introduction

Sequencing

A distributed computing environment consists of a set of components, typically assumed to operate asynchronously, which share and coordinate their efforts toward executing a computation. Such sharing requires the coordinated maintenance of an information base so that the sequencing decisions made by the components are consistent with the specifications for the computation. To do so, these components exchange the information required to support the decision process involved in coordinating their efforts. Such a decision process may be modeled as a function **f** which when applied to the state **s** of the computation, returns a sequencing decision **d**; i.e., if **S** is the set of all possible states and **D** is the set of all possible decisions, then **f:S->D**.

A consequence of asynchrony is that the clocks of these components can never be in perfect agreement. Yet, if the components are to be mutually consistent in their decision-making, they must maintain consistent views of the computation's state **s**. The maintenance of consistent views of state information upon which sequencing decisions can be made is a critical problem of the management of distributed computations. The choice of decision

function **f** and the exchange of information which leads to the establishment of its domain **S**, will be referred to as the "sequencing protocol".

Optimism, Pessimism and Balance

There are three general classes of sequencing protocols. One class guarantees *a priori* that all sequencing decisions are consistent with the specification of the computation. This class of protocols is termed pessimistic, since decisions are postponed until all relevant information is available and consistent. A hypothetical global observer who can observe events as they occur may notice inefficiencies such as those caused by the computing components waiting because they do not yet have the necessary information to enable them to go ahead. Thus, when a decision needs to be made, the pessimistic approaches incur an overhead (delay etc.) associable with determining **S**. The second class of protocols involves making favorable assumptions about all unavailable information required to proceed with the computation. This class is termed optimistic since decisions are never postponed because relevant information is unavailable. Instead, if such optimistic actions result in a deviation from specifications, such a deviation is detected and compensating actions to restore consistency with specifications are taken. The optimistic approaches incur an overhead for detection of and recovery from inconsistencies.

There is some controversy in the literature as to the circumstances under which each protocol class yields superior performance. However, there is no reason to believe that either extreme protocol is optimal in any particular

circumstance. There is a third class of protocols which introduces quantifiable and variable measures of optimism into a previously pessimistic protocol to obtain a **balanced** sequencing protocol. This third class of protocols, which constitutes the spectrum of strategies spanning the extremes, is the subject of this dissertation. It is the thesis of this dissertation that such a protocol will, in many circumstances, yield better performance than either completely pessimistic or completely optimistic sequencing protocols.

While optimism can in general be introduced into any type of computation, it has been studied most extensively in the contexts of database concurrency management and distributed simulation. In both contexts, protocols aimed at combining the best of both approaches by tempering the level of optimism have been proposed. [KUN81] presents a database protocol which optimistically reads the database, then pessimistically updates it. The "five-color" locking protocol of [DAS90] attempts to improve upon two-phase locking by admitting some serializable non-two-phase schedules. It requires a validation component because it admits some non-serializable schedules. Performance data for this protocol is not currently available. [FUJ88] obtains mixed results for simulation strategies which "look ahead" by estimating the simulated time of future events.

Outline

Chapter 2 covers the background in distributed computing and sequencing needed to study this document.

Chapter 3 demonstrates the universal applicability of the concept of balanced sequencing by presenting the design of a very general sequencer capable of sequencing any program at any desired level of optimism. The operation of such a sequencer is illustrated with examples. However, the high computational cost of such a general approach suggests the need for specialized balanced sequencing protocols targeting narrower problem domains.

Chapter 4 presents the design and performance analysis of some specialized balanced sequencing protocols. The first problem considered is the Producer-Consumer problem. Even for this simple, extensively studied problem, it is demonstrated analytically that, depending on the relative lengths of the operations "produce", "consume", "read shared memory" and "write shared memory", the optimal balance can be any where in the spectrum between extreme optimism and pessimism.

Next, in Chapter 4, the balanced version of a distributed implementation of a semaphore is presented. An analytical relationship is established which relates the optimal balance to characteristics of the system and the application served by the semaphore. Again, it is demonstrated that the optimum can lie any where in the spectrum of possibilities.

The final section of Chapter 4 adapts the design and performance analysis of distributed semaphores to distributed locking. This adapted analysis is then

applied to the data used in a prior simulation study of distributed concurrency management [CAR88].

In Chapter 5, a previously unknown pitfall of optimistic sequencing which can lead to severe and permanent degradation of performance with the passage of time is identified. For some simple networks of processes employing the Time-Warp optimistic protocol, the extent of degradation is quantified as a function of the speed at which processes progress and recover. Finally, a solution to this problem is proposed, which works by varying the level of optimism and employs Dijkstra's "Diffusing Computation" paradigm.

Chapter 6 is a literature survey. Here, some of the key contributions to this dissertation from existing literature are summarized.

Chapter 7 summarizes the results and conclusions and suggests directions for future work.

Chapter 2
Background

Distributed computing

The concepts of a process and a distributed system

"Distributed Computing" is a study of the interaction of concurrently active processes which do not share a common sense of time; i.e., processes which are not "centrally clocked". Each process in its most general form, consists of a "local state" and a program which operates on that state and communicates with other processes. The local state of a process is the set of state variables which only that process can access. A set of processes, together with the "protocols" which govern their interaction constitute a distributed system.

Shared state Vs. messages

The processes of a distributed system can communicate either by their use of shared state, or by exchanging messages. The choice of communication mechanisms can be further resolved based on assumptions about them. A popular flavor of shared state, called **Concurrent Read, Exclusive Write** or CREW, assumes that while several processes can concurrently read the value of a shared variable, a process seeking to modify the value must have exclusive access. In this dissertation, the following properties are assumed for message-based communication mechanisms:

- Processes are inter-connected by channels.
- Each channel is matched with exactly one process which can send messages on it and exactly one process which can receive messages on it.
- A channel neither loses messages nor generates them of its own volition.
- A channel delivers messages in the order in which they are sent, after an arbitrary, unbounded delay.
- The capacity of a channel is unbounded.

The above is by no means the only possible model of communication media. However, it is the most common. The Synchronous Communication model of CSP [HOA85] is an example of an alternative model.

A general introduction to sequencing

Guarded Commands

The "guarded command" denotation of programs has proven useful in several contexts of distributed computing research. For our purpose, it provides the basis for an elegant definition of the level of optimism in a sequencing protocol, and hence, an elegant definition of balanced sequencing.

A guarded command denotation of a program has the form: **do** G_0 —> S_0 G_1 —> S_1 ..... **od.**

The G_i are predicates on the variables of the program and are called guards. The S_i are steps which modify the program's variables. The interpretation is that a step may be executed any time its guard is true, and the program terminates when all guards are false [DIJ76]. It is assumed that the values of a program's variables– and hence the truth of its guards– can only be changed by the execution of a step. Therefore, a true guard remains true and a false guard remains false at least until the execution of the next step. Determining a true guard and executing the corresponding step is atomic.

Processes of a distributed program are disjoint subsets of guard-statement pairs. The need for communication arises when more than one process accesses any given variable. A different model of a distributed computation and a notion of correctness called "sequential consistency" are described in [LAM79].

Pessimism, Optimism and Balance

When a process knows enough about the program's state to determine the truth or falsehood of a guard G, it may accordingly determine whether or not to execute the corresponding statement S. It is when a process can not determine the truth of G that optimism enters the picture. A pessimistic sequencing protocol would disallow the execution of S and an optimistic protocol would allow it. A continuum of balanced protocols are obtained by associating with each guard G, a predicate G_p such that $G \Rightarrow G_p$. **Under a balanced protocol, when a process can not determine the truth of G, it**

allows S to execute if and only if it can determine that G_p is true. In particular, choosing G_p = G yields the pessimistic end-point and choosing G_p = true yields the optimistic end-point.

Validation and Recovery

An optimistic or balanced protocol must include some means to determine when an invalid sequencing of events has occurred, and to compensate for such occurrences.

Validation methods typically involve a compromise between the desire to lower the cost of determining the validity of an execution and the undesirable possibility of flagging some valid executions as invalid. [PAP86] presents a discussion of such trade-offs for database concurrency control. [JEF85] and [KUN81] use only the time-ordering of events for validation. The protocol presented in the next chapter validates by comparing run-time dependences between events of a program's execution with the dependences specified between the steps of the program. If, for example, correctness were specified as a predicate on the observable result of executing a program, all of the above approaches have the potential to flag valid executions as invalid.

Proposed approaches to recovery can be classified as **backward** and **forward**. Most proposed recovery protocols in the literature [WOO80,

STR85, KOO87, JOH88, JEF85, BHG87][1] are **backward** recovery protocols, as is the protocol proposed in the next chapter. In all of these protocols, a state which resulted from a valid prefix of the computation is restored, and the computation resumes from that state, *thus ensuring that the result of the computation is in no way affected by the incorrect sequencing which occurred.* Two possible weaknesses of **backward** recovery are the cost of saving intermediate states and the potential for **cascading rollback** (A phenomenon whereby recovery in one process or transaction necessitates recovery in others and so on. Such a cascade can repeatedly affect a process, in the limit causing it to roll back to its initial state). [RUS80] proved that for a system of processes communicating over FIFO channels, given the ordering of events defined in [LAM78], it is necessary and sufficient to save the state of a process immediately prior to sending each message to avoid cascading rollbacks.

Forward recovery consists of performing additional actions to ensure that a correct result is obtained despite the invalid sequencing in the past. Such a protocol is proposed in [GAR83]. [KOR90] proposes a theory for forward recovery in transaction-based systems based on the concept of compensating

---

[1]Text books (such as [BGH87]) are referenced several times in this document. In every case, the referenced text is one example of several texts on that subject which could have been referenced in that context.

transactions. *These protocols guarantee the restoration of the semantic integrity of the computation. They may, however, produce results which can not occur in the absence of incorrect sequencing and the subsequent recovery.* The need to save intermediate states and the cascading rollback problem are avoided with forward recovery. Also, the resulting improvement in the efficiency of recovery can significantly improve the gains from balanced sequencing, as is illustrated by the second and third example studied in chapter 4. However, such protocols are based on the semantics of the applications they serve; i.e., no generally applicable protocol has been proposed.

Chapter 3
A General-purpose Sequencing Protocol

Motivation

In the following paragraphs, a general protocol capable of executing any given program at any specified level of optimism will be presented. Inevitably, such a general protocol will be less efficient than its specialized counterparts such as those described in the next chapter. The anticipated application of this general approach is to investigate the effectiveness of balanced sequencing for a given application before one undertakes the non-trivial task of devising a specialized sequencing protocol for that application. It may also serve to guide the design of specialized protocols for complex applications.

The Detailed Model

The model consists of four component protocols and the three data structures they use. The data structures are the **program** which specifies the computation, the **log** which denotes the history of the computation and the **sequencing options table** abbreviated **SOT** which denotes valid possibilities for the computation's progression. The component protocols of the general sequencing protocol are **triggering**, initiating actions, **logging**, recording the computation's history, **validation**, checking the consistency of

the log with the program, and **recovery**, eliminating any inconsistency discovered by the validation protocol.

The Data Structures

The **program** is denoted by a directed graph. Each node is labeled with a quadruple **<n, s, I, O>**. **n** is a unique identifier. **s** is a step of the program. **I** is a predicate denoting the dependence of **s** on other steps and **O** is a predicate denoting the dependence of other steps on **s**. The attribute **s** is of the canonical form **U := f(V)** where **U** and **V** are sets of variables termed the **modification domain** and the **invocation domain** respectively. **I** is a predicate of the canonical form **one_of**(*a set of sets of dependences*). **O** is a predicate of the canonical form **one_of**(*a set of pairs in which the first element of each pair is a predicate on the state of the data and the other is a set of dependences*). The boolean function **one_of** is true iff precisely one of the elements of its argument set contains only satisfied dependences, any satisfied dependence belonging to other sets of its argument also belong to that set and the predicate if any associated with that set of dependences is true. Each directed arc represents a dependence of its destination on its source and is labeled with a triple **<t, a, d>**. **a** is a unique identifier and **d** is a datum of the computation. **t** is the type of the dependence, one of:

- **MM**, signifying that **d** is an element of **U** of the source and the destination nodes,

- The predicate O for this node is **one_of**({X>Y, 6}, X=Y, 7, 8}, X<Y, 9, 10}). Interpretation:— Upon executing this node, the dependences which become "enabled" are **6 if X>Y**, **7** and **8 if X=Y** and **9** and **10 if X<Y**.

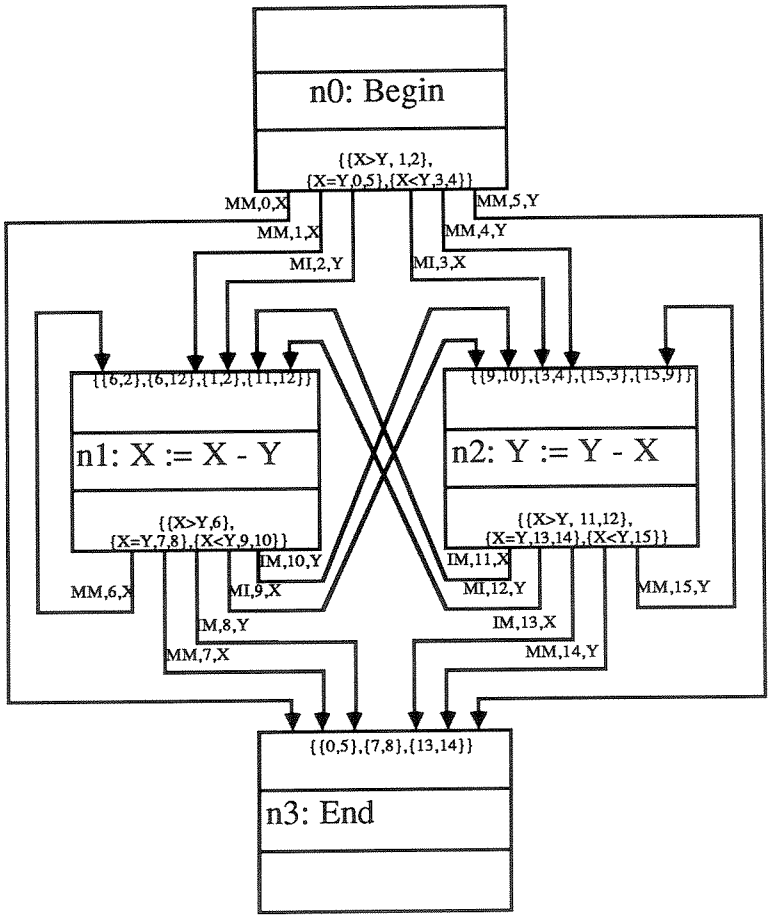

Figure 1

The **Log** of a computation is a record of the events of that computation and the dependences among them. The log, like the program, is denoted by a directed graph. The nodes of this graph are events (executions of individual program steps) and the arcs represent dependences among them. Each node of the graph is a triple, **<n, E, C>**. **n** identifies the step in the program which was executed. **E** is the exit condition; i.e., the dependence of other events on it. This is the set of all valid combinations of out-going arcs from the node. **C** is a checkpoint; i.e., a record of the version of each datum used by the event. A datum may be checkpointed either implicitly by recording its version number or explicitly by recording its version number and its value. This definition of a checkpoint differs from the usual meaning of this term in literature related to crash recovery for databases and general distributed computing [BER87, CHA72, CHA75, RUS80, WOO80, KOO87, JOH88]. A checkpoint usually refers to a state of a process or an entire system which can be restored if such restoration is necessitated by any future event such as a crash. The **C**s are refered to as checkpoints because collectively, they serve the same purpose as traditional checkpoints; i.e., they represent previous states of the system. The logging of an event is **complete** if all of its logical predecessors are logged. A set of completely logged events constitutes a **complete segment** of a log and the set of all completely logged events is the log's **maximal complete segment**.

The **SOT** denotes valid ways in which the computation may progress. It consists of two types of entries. An **I_Entry** is the **I** attribute of an initiated step with each dependence tagged with the initial version number of the datum. An **O_Entry** is computed from the **O** attribute of a terminated step as follows:

- Remove the **<predicate, dependence set>** pairs whose **predicate** component evaluates to **false.**
- Remove the **predicate** component of the remaining **predicate, dependence set>** pairs.
- Tag each dependence with the version of the datum it represents.

The initiation of an event results in the creation of an **I_Entry** in **SOT.** The termination of an event results in the creation of an **O_Entry** in **SOT.**

Note that the initiation and termination of events are recorded in both the **SOT** and the **log**. In fact, the **SOT** can be computed from the **log**, and hence, the **log** can serve the above role of the **SOT**. This redundancy serves to improve efficiency in two ways:

- Since the log serves as the basis for validation, an entry remains in the log at least until the event is validated. However, entries in SOT can be simplified and/or removed by a **reduction** process (described later) without regard to validation.

- The organization of information for efficient validation (the log) vs. efficient triggering (the SOT) is different. Entries in the log are arranged in the partial order determined by their dependence on other events, since the proposed validation protocol analyzes these entries in some order consistent with this partial order. For the SOT, it is advantageous to organize entries according to the corresponding nodes of the program graph, since, to determine whether a node in the program graph should be triggered, only O-entries of its predecessors in the program graph are analyzed.

The Protocol Components

The **triggering** protocol is the component which initiates events. The pessimistic aspect of a triggering mechanism is the overhead that goes into reducing or eliminating the possibility of triggering events out of sequence. The optimistic aspect is the triggering of events before sequencing errors can be ruled out.

The basic strategy consists of maintaining the **SOT**. The manner in which this data structure is distributed and/or replicated is an independent issue.

Initially, **SOT** has no entries. The initiation and termination of events result in the addition of **I_** and **O_ entries** respectively to **SOT**. Let **q** stand for the predicate **conjunct (O_Entries) and not disjunct (I_Entries)**. The decision to trigger an event is correct if **q** implies its **I** attribute, wrong if **q**

implies the negation of its **I** attribute and otherwise undecidable. It is in the treatment of the undecidable instance that the issue of optimism enters the picture. A pessimistic strategy will treat this as wrong and an optimistic strategy will treat this as correct. To implement a balanced strategy, **Ip** is chosen for each step such that **I** implies **Ip.** (**q** implies **Ip**) becomes the criterion for triggering the event. The spectrum of protocols ranging from pessimistic to optimistic is achieved through the choice of appropriate **Ip**. Pessimistic triggering is achieved by choosing **Ip = I** while optimistic triggering is achieved by choosing **Ip = true.** In general, weakening **Ip** increases optimism and reduces pessimism and vice versa, thereby providing a full spectrum of protocols. Any **Ip** may be strengthened in an arbitrary manner without jeopardizing any safety property, although liveness can be jeopardized.

If the creation of **I_** and **O_** entries are the only modifying operations on **SOT**, its information content will be identical to that of the computation's log (discussed later). The **SOT** grows monotonically with the progress of the computation, and hence, so does the cost of using it in triggering decisions. Therefore, means for **reducing** the entries in **SOT** without altering the available sequencing options are desired. Such a **reduction** is effected by removing all dependences tagged with old version numbers and then removing all entries which contain a null set of dependences. From the perspective of minimizing the cost of triggering decisions, it is desirable to reduce the **SOT**

every time a new version of any datum is created. However, the reduction itself is an overhead. One way to balance the incurrence of these overheads is to reduce the **SOT** whenever some pre-determined number of modifications to data have occurred.

The **Logging** protocol maintains the log of the computation. At the termination of each event, a node is added to the log. All attributes of the node, **n**, **E** and **C** are available upon completion of the event. However, if any optimism is involved in the triggering of events, the situation could arise wherein the log contains an event but not all of its logical predecessors (The event is not completely logged). In such a situation, information necessary to compute the set of incoming arcs may not be available at the time of logging an event. Since the incoming arcs can be derived from the **C** attributes of a complete segment of the log, the task of filling in the incoming arcs must be postponed until the node achieves completeness. Nodes may be removed from the log either as a part of the process of recovering from a fault or via garbage collection. A node is a candidate for garbage collection if all its predecessors are candidates for garbage collection, it represents a correctly sequenced event and all the data it explicitly checkpoints are explicitly checkpointed by one or more of its successors.

The **validation** protocol determines the consistency of the log with the program and identifies any events that may be incorrectly sequenced. An event **n** in a complete segment of a log is in sequence if its set of incoming arcs

satisfies the **I** attribute of node **n** of the program and its set of outgoing arcs satisfies its **E** attribute. The first step in the validation procedure is to compute the arcs among the nodes to be validated from the **C** attributes of the logged events. (For each datum modified by **n**, an in-coming arc either from each event that invoked the initial version of that datum, or, if there are none, an in-coming arc from the event that created the initial version. For each datum invoked by **n**, an in-coming arc from the event that created that version of that datum.) Then, for each of the nodes, the **n** attribute identifies the corresponding step of the program, and consistency in sequencing can be verified. 1) The set of in-coming arcs must be consistent with the **I** attribute of node **n** of the program and 2) the set of out-going arcs must be consistent with the **E** attribute of the event. The nodes which don't meet condition 1 represent events executed out of sequence. For nodes which don't meet condition 2, the successors whose removal restores condition 2 represent events executed out of sequence. The set of events determined to be executed out of sequence is the **fault.** The domain of a fault is the data whose state is affected by the fault; i.e., the modification domain of the fault and its successor events.

The **recovery** protocol negates the effect of the detected fault. Approaches to recovery are classified as **forward** if they involve including additional events in the execution to compensate for faulty events and **backward** if they involve state restoration and resumption of computation at

or prior to the origin of the fault. (Under an alternative classification, recovery protocols which retry a computation are classified as **forward** and others, whether dependent on or independent of semantics, are classified as backward.) This research restricts itself to **backward** strategies since they are independent of the semantics of the program's steps. Backward recovery involves a sequence of three steps, **restoration, reconstruction** and **resumption.**

State **restoration** is achieved by a backward search of the log starting from the fault until an explicit check-point for each element of the fault domain is encountered. (Note that **SOT** should also be restored. This is achieved by treating the entries in **SOT** as data objects for the purposes of checkpointing and recovery.) Let's call this set of explicit checkpoints the **anchor**.

State **reconstruction** is optional. By repeating the events logged between the anchor and the fault, the state of the fault's domain just prior to the fault is reconstructed. At this point, there is at least enough information to sequence the next event correctly, and hence a positive rate of progress is guaranteed regardless of the amount of optimism and regardless of the checkpointing intervals if mis-sequencing is the only type of fault encountered.

The computation may resume either from the restored state or from the reconstructed state. The primary advantage of incorporating the reconstruction

step is that it is possible to guarantee progress of the computation without having to checkpoint every state of every variable.

Consider the execution of the GCD program under an optimistic protocol. Figure 2(a) shows the log of an incorrectly sequenced execution of this program. In this case, n2:2's dependence on n1:2 is inconsistent with the exit condition of n1:2. For this specific fault, there is an obvious forward recovery fix; i.e., extend the execution with the event Y := Y + X. Clearly, this fix is based on the semantics of the event which is out of sequence. This is in general true of any forward recovery scheme and hence there is no such thing as a general purpose forward recovery scheme for any computation based on any program.

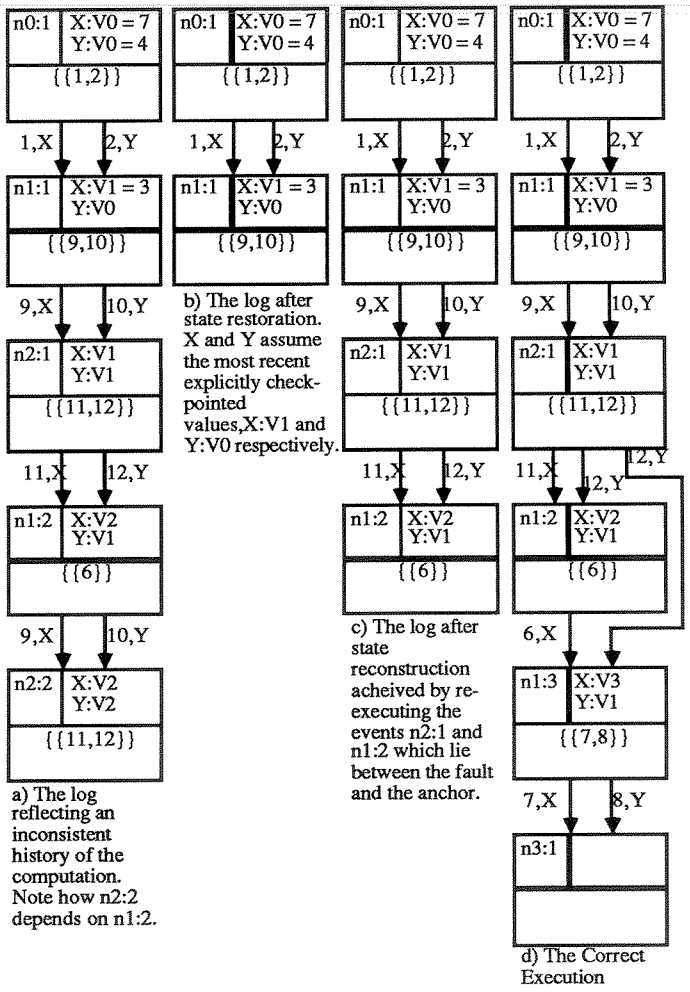

Figure 2

The first step is to restore the values of X and Y to a previous consistent state. The most recent explicit checkpoints are X:V1 at event n1:1 and Y:V0 at event n0:1. If there were any intervening events between n0:1 and n1:1 which

modified Y, it would be necessary to find an earlier checkpoint of X. This in turn could necessitate finding an earlier checkpoint of Y an so on. This is the cascading rollback or domino effect discussed in [WOO80] and [RUS80]. In this case, there is no cascading rollback, and the log after state restoration looks like figure 2(b).

One may return to the computation after state restoration, but this approach has one serious problem. There is no reason why the same problem can not recur. Hence, conceivably, the computation could thrash indefinitely. State reconstruction avoids this problem regardless of the frequency of explicit checkpoints. State reconstruction consists of re-executing the log between the anchor and the fault; i.e., the events n2:1 and n1:2, thereby **reconstructing** the state just prior to the fault before returning to optimistic sequencing (figure 2(c)). The incorrect optimistic choice which necessitated this rollback is now known to be incorrect, (because otherwise, rollback would not have been initiated) and will therefore not be repeated. However, a different incorrect choice can be made at this time. Indefinite thrashing can not occur because:

- There are finitely many steps in the program.
- Therefore, there are finitely many incorrect choices available.
- No incorrect choice will be made more than once.

Figure 2(d) shows the log of the correct execution.

Evidence of Generality

This section describes and exemplifies a methodology for modelling distributed computing environments and the consistency management protocols they employ in the form described in the previous section.

The General Strategy

- Identify the synchronous components of the environment. Examples are processes of a system of communicating processes, transactions of a transaction based system, the individual programs of a multi-programming environment, actors of a dataflow computation etc.

- Represent the role of each synchronous component as a fragment of a program.

- Determine (from the integrity constraints) the manner in which these components may correctly interact.

- Integrate the program fragments using information determined in the previous step.

- Mimic the behavior of the consistency management protocol to be modeled by determining the appropriate **Ip** for each step of the program, the control mechanism for choosing among steps when more than one is enabled by a valid **Ip**, the criterion by which the validation procedure determines inconsistencies etc.

An Example

Consider two concurrent transactions, T1 and T2 in a distributed database constrained to execute serializably. Between them, the transactions access three data, P, Q and R. Figure 3. is a program representing the correct concurrent execution of T1 and T2; i.e., the set of all serializable interleavings of their actions. If T1 and T2 had to be denoted as sequential programs, they may be written:

T1: read Q; read R; write P as f1(Q,R).

T2: read P; write Q as f2(P).

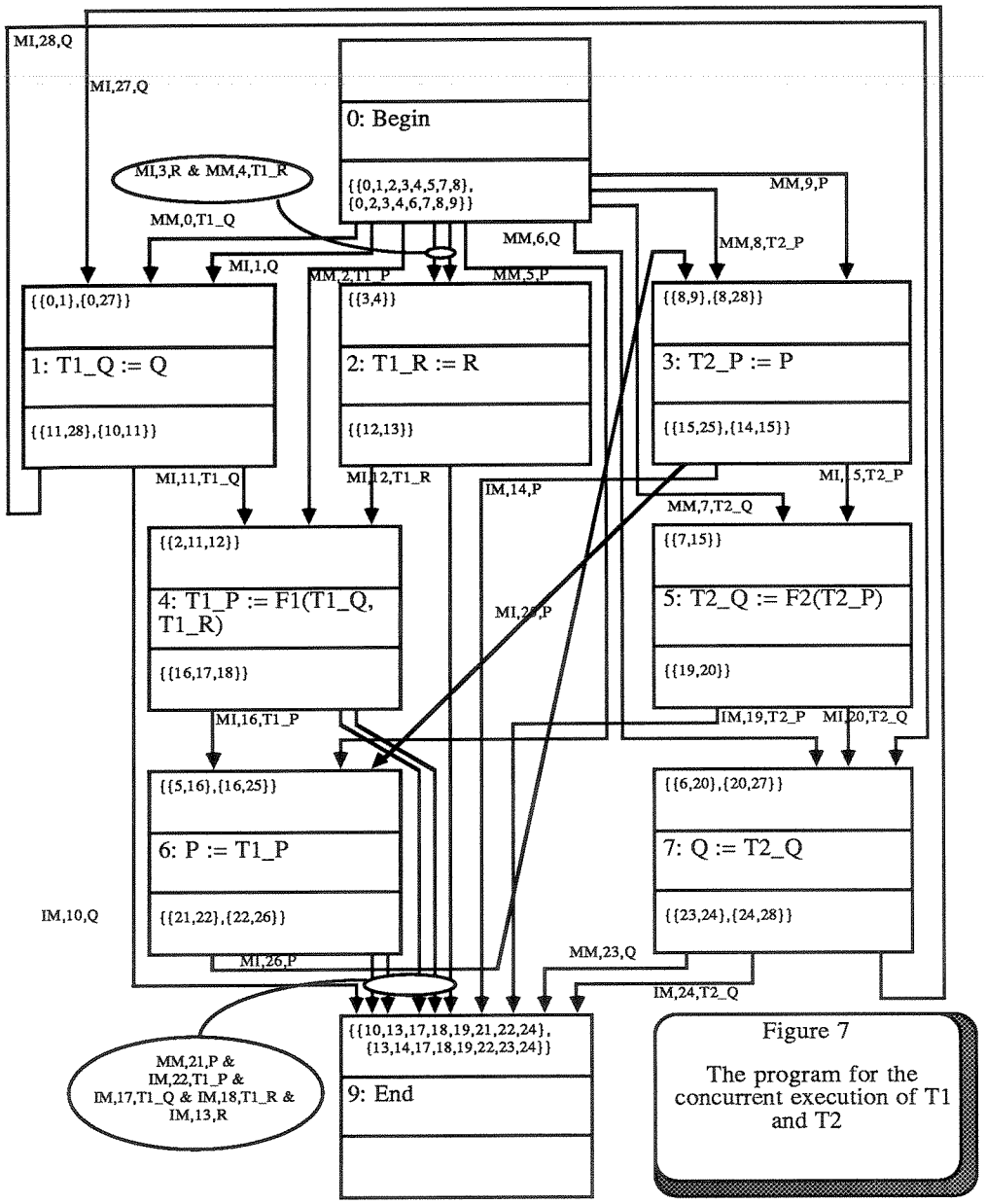

Figure 7

The program for the concurrent execution of T1 and T2

Figure 3

The following sections show how four "traditional" protocols to sequence the events of T1 and T2 are special cases of the general purpose protocol. Details of these protocols can be found in chapter 6 and the referenced papers. T1 and T2 are the two processes. For the purpose of this example, a dependence is called **external** if it derives from the use of a shared variable (here, P, Q and R) and **internal** otherwise.

Schneider's Protocol Extended to Allow Non-deterministic Decisions

This is a strictly pessimistic strategy. The triggering criterion **Ip** equals **I**. An extension of **Schneider's protocol** is used by the control mechanism to resolve non-deterministic choice. *(for example, initially, either nodes 1 and 2 or node 3 can be triggered. T1 will broadcast a request to initiate nodes 1 and 2 and T2 will broadcast a request to initiate node 3. The request with the earlier time-stamp will win.)*

Two-phase Locking

This is frequently called a (sometimes, **the**) pessimistic protocol, but in our framework it must be classified as pessimistic WRT internal dependences and balanced (neither strictly optimistic nor strictly pessimistic) WRT external dependences (and hence is an instance of a balanced protocol with the internal dependences constituting the boundary of optimism). The triggering protocol employs a variable for each shared (lockable) object (P_lock, Q_lock and R_lock) which can take on the values T1, T2 or UNLOCKED. The triggering

criterion **Ip** equals (**I** with external dependences removed) AND (the transaction to which the step belongs has locked all objects represented by its external dependences). For example, $\mathbf{Ip}_1$ = (one_of {{0}}) AND (Q_lock = T1). The control strategy is:

- Initially, all locks are UNLOCKED.
- When a step requires only one or more locks to satisfy its Ip, a lock request is broadcast by that transaction. For example, when **one_of {{0}}** is true, **"Request Q_lock"** is broadcast by T1.
- Locks are granted in time-stamp order of their requests using Schneider's protocol. The granting of a lock need not be broadcast since Schneider's protocol guarantees that all transactions see the same picture.
- Locks are relinquished when a terminated step has an external output dependence and the transaction will not subsequently request locks. For example, because of arcs 10 and 28 of step 1, T1 broadcasts **"Relinquish Q_lock"** any time after broadcasting **"request P_lock"** when node 6 comes up for execution.

Commit protocol (Kung & Robinson)

This is frequently called a (sometimes, **the**) optimistic protocol, but in our framework it must be classified as pessimistic WRT internal dependences and external dependences representing variables in the modification domain and optimistic WRT external dependences in the invocation domain (and hence is

an instance of an optimistic protocol with the internal dependences and external dependences representing variables in the modification domain constituting the boundary of optimism. The triggering criterion **Ip** equals (**I** with external dependences not representing variables in the modification domain removed).

Jefferson's "virtual time" protocol

This is another approach to "optimistic" sequencing. The triggering criterion $\mathbf{I}_P$ equals (**I** with all external dependences removed). This approach contrasts with that of Schneider in that transactions "optimistically" presume that protocol related messages are received in time-stamp order and hence, the set of all received messages and the assumption that there are none in transit with intermediate time-stamps form the basis for determining the truth of the pre-conditions of the production rules. The receipt of a protocol-related message out of turn is the criterion used by the validation protocol to determine that an inconsistent state exists.

Chapter 4
Some Specialized Balanced Sequencing Protocols

The previous chapter discussed the design of a general-purpose protocol which can be used to sequence any given program at any desired level of optimism. Unfortunately, this generality comes at a potentially steep price. Consider, for example, the maintenance of the data-structure **SOT** in the general-purpose approach when the program is executing on a distributed system with disjoint sub-graphs of the program-graph implemented as processes and arcs connecting these sub-graphs implemented as channels. In the worst case, a process must communicate with each of its successors for every step it executes, no matter what the chosen level of optimism. Further, in the worst case, a process must communicate with each of its predecessors either when triggering a step or when validating it. This communication can prove to be quite expensive, and can negate any performance gains obtained by the introduction of optimism. The remainder of this section addresses this problem by presenting and analyzing a series of specialized balanced implementations of common distributed computing primitives and paradigms. In each of these examples, the level of optimism can be picked from a spectrum of choices ranging from optimistic to pessimistic by varying a parameter of the

protocol. The analysis determines the optimal level of optimism as a function of the characteristics of the distributed system.

It is a folk-theorem of distributed computing that every synchronization problem can be reduced to some combination of two basic problems: the producer-consumer problem, and the mutual exclusion problem [LAM83]. Of these, the producer-consumer problem is considered to be "easy", and the mutual exclusion problem is considered to be "difficult".

This chapter presents specialized balanced sequencing protocols and demonstrate their applicability for a series of examples. The first of these is the "easy" producer-consumer problem. The second example takes the most efficient known pessimistic distributed algorithm for mutual exclusion in message-based systems [MAE85], adapts it to implement a distributed semaphore, generalizes the algorithm for balanced sequencing and studies the relationship between a system's characteristics and the optimal balance between optimism and pessimism. The third example adapts the work done for the second example to the domain of distributed locking.

The Producer-Consumer Problem

This example considers the sequencing of an asynchronous producer-consumer system sharing a buffer of size 1. The producer-consumer system may be informally specified as follows:

- **MI**, signifying that **d** is an element of **U** of the source node and an element of **V** of the destination node,

- **IM**, signifying that **d** is an element of **V** of the source node and an element of **U** of the destination node.

A datum is defined as a triple **<name, value, version number>** where **name** is a unique identifier which does not change, **value** is the state of the datum and can change and **version number** is a counter of the changes undergone by **value**. Alternately, a datum is an association of a value with its name for each version number from some initial value, say 0, up to its current value.

Figure 1 is a program implementing Euclid's algorithm for computing the Greatest Common Denominator of two positive integers.

- The variables of this program are the integers **X** and **Y**.

- The step **s** associated with node **n1** is **X := X - Y**.

- The predicate **I** for this node is **one_of**({6, 2}, {6, 12}, {1, 2}, {1, 12}). Interpretation:— this node can be executed only if either dependences **6 and 2** or **6 and 12** or **1 and 2** or **1 and 12** are "enabled".

- The **producer** repeatedly "produces" data and writes it to the shared buffer.
- The **consumer** repeatedly reads data from the shared buffer and "consumes" it.
- **integrity constraint:** All produced data must be eventually consumed in the order in which they are produced.
- **optimality criterion:** Maximize the average rate of progress, where progress is measured as the number of data correctly produced and consumed.

Variables of the optimality analysis are the processing times for the four key operations, **produce**, **consume**, **read-buffer** and **write-buffer**. Because these quantities are assumed to be invariant over time, the optimal protocol does not result in any timing faults, and hence, the cost of recovery does not play any role in the analysis. The analysis also does not penalize the non-pessimistic protocols for the additional memory they use.

The specification for this example has been deliberately chosen to be as simple as possible in order that the complexity of the analysis not obscure the main point of the example; i.e., that balanced protocols can be a viable options even in such a simple system. All of the above factors, namely, variations in processing time, the cost of recovery and the cost of memory, can be

significant in a "real-world" application. The subsequent examples use more realistic but still analytically tractable specifications.

A pessimistic solution

The following program is a conventional pessimistic protocol for the Producer-Consumer problem. The producer ensures that a produced datum has been fetched by a consumer before over-writing it.

```
System pessimistic;

 Process producer;
  Begin
   produce (item);
   Repeat
    write item to buffer and signal "buffer full";
    produce (item);
    await "buffer empty"
   Forever
  End;

 Process consumer;
  Begin
   Repeat
    await "buffer full" and read item from buffer;
    consume (item)
```

```
    Forever
  End;
```

A Balanced Sequencing Solution

The following solution to this problem employs a phased approach. In each phase, a producer optimistically produces and writes several items to the buffer without waiting to ensure that old items have been correctly consumed. At the phase-boundary, the producer checks with the consumer to see if all has gone well. If so, the next phase begins. Else, corrective action is taken, and then the next phase begins. The measure of optimism is the size of a phase; i.e., the value of N in the program below.

```
System optimistic;

 Process producer;
  Begin
   await "begin producing";
   Repeat
    For i := 0 to N - 1 Do Begin
     produce (item[i]);
     write item[i] to buffer and signal "item[i]
written"
    End;
    await "end phase" and read failure_at from
consumer_status;
```

```
{if item[x] was over-written before the consumer
could read it, then failure_at = x.  If the consumer
could fetch all items, then failure_at = N, and the
following for-loop will be skipped.}
     For i := failure_at to N - 1 Do Begin {recover
from failure if any}
      write item[i] to buffer and signal "buffer
full";
      await "buffer empty"
     End
    Forever
  End;
 Process consumer;
  Begin
   signal "begin producing";
   Repeat
    item# := 0;
    Repeat
     await "item[i] : i ≥ item# written" and read
item;
     If i = item# Then Begin {no failure this
time}
       consume (item);
```

```
        item# := (item#+1)
      End
    until (item# = N) {phase completed without
failures} or (i > item#); {there was a failure}
    write item# to consumer_status and signal "end
phase";
    For item# := item# to N - 1 Do Begin {recover
from failure if any}
      await "buffer full" and read item from
buffer;
      signal "buffer empty"
    End
  Forever
End;
```

Performance Analysis

This section presents a derivation of the optimum balance in terms of the speeds of the **produce**, **consume**, **shared read** and **shared write** operations.

Assumptions

- Processes interact only through their use of shared variables. In particular, the "write and signal" statement is implemented by a write-access to shared memory and the "await and read" statement is implemented by a sequence

of one or more read-accesses to shared memory which terminates when the awaited condition is detected.

- All operations other than a **read** or **write** access of shared memory, **produce** and **consume** take negligible time in comparison, and can be ignored in the analysis.

- If a read and a write to a shared variable overlap, the write is not affected in any way. The read fails, but takes the same amount of time as a successful read.

- A concurrent read or write takes the same time as a read or write to a single variable.

Notation

N denotes the number of **produce** operations in each phase, and serves as the measure of optimism. The subscripts **p**, **c**, **w** and **r** denote the operations **produce**, **consume**, **write** to shared memory and **read** from shared memory respectively. **T**, **P** and **C** with appropriate subscripts denote **time**, actions of the **producer** and actions of the **consumer** respectively. For example, $\mathbf{T_p}$ denotes the time taken by a **produce** action and $\mathbf{P_w}$ denotes a **write** action taken by the **producer**.

Analysis

Timing diagrams are employed in this analysis to depict the behavior of a producer and a consumer during an optimistic phase. The producer repeatedly

produces an item and writes it to the buffer. The consumer repeatedly attempts to read a buffer until it succeeds— i.e., the read does not overlap a producer's write and the required item is read— then consumes it.

The first step is to establish the domain over which any protocol employing optimism can not be guaranteed to perform better than the pessimistic protocol. Intuitively, if **produce** operations are fast WRT **read** operations, optimistic approaches are unlikely to be suitable because an optimistic producer would begin to overwrite the buffer before the consumer had a reasonable chance to read it. This intuition is supported by figure 4, which is a timing analysis of:

condition 1:— $T_p < 2T_r$

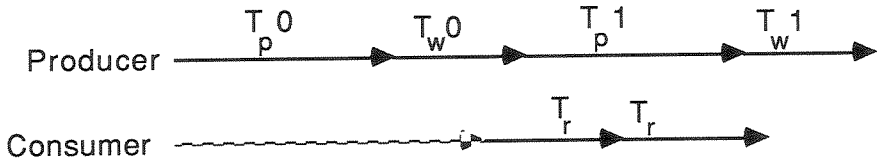

Figure 4

Here, the optimistic producer writes item[0] to buffer in time frame T_w0, produces item[1] in time frame T_p1 and writes it to buffer in time frame T_w1. Two successive attempts by the consumer to read item[0] fail, the first because its time frame overlaps the end of T_w0 and the second because its time frame

overlaps the beginning of T_w1. Subsequently, item[0] can not be read because the producer has already begun to overwrite item[0] with item[1].

It follows from figure 4 that if any optimistic protocol is employed when condition 1 holds, the consumer may miss the very first production, thereby necessitating the repetition of the entire phase. Therefore, under condition 1, the pessimistic protocol is optimal.

Figure 5 is a timing analysis of:

condition 2:- $T_p \geq 2T_r$ and $T_w + T_p \geq T_c + T_r$

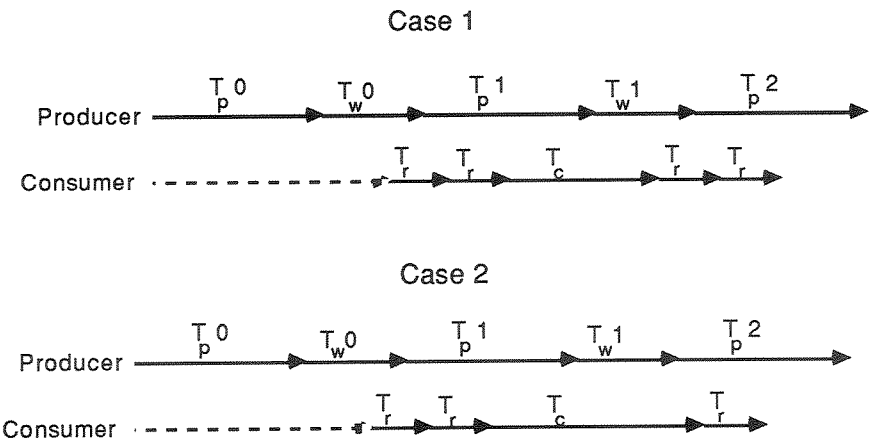

Figure 5

In case 1, $T_p + T_w \geq T_c + 2T_r$. In case 2, $T_c + 2T_r \geq T_p + T_w \geq T_c + T_r$. The timing analysis demonstrates that in either case, if an item is read by the

consumer within time $2T_r$ after it is written to the buffer, it will be consumed no later than time T_r after the next item is written to buffer. This in turn guarantees that the next item will be read within time $2T_r$ after it is written to the buffer. Since the use of the "begin producing" signal at start-up ensures that the first item produced will be read within time $2T_r$ after it is written to the buffer, a produced item will not be missed regardless of the chosen level of optimism. Therefore, under condition 2 unbounded optimism is optimal.

Figures 6 a and b are timing analyses of:

condition 3:- $(T_p \geq 2T_r)$ and $(T_w + T_p < T_c + T_r)$

As before, at start-up, the "begin producing" signal ensures that the consumer reads the first item within $2T_r$ after it is written to buffer. Thus, the lag between the completion of the first write to buffer during an optimistic phase and the completion of its successful read (referred to as "the lag" in subsequent paragraphs) is at most $2T_r$. Figure 6a demonstrates that the start of the last successful consume may lag the completion of the corresponding write by at most T_p. Therefore, during the remaining N-1 produce-consume cycles of an error-free optimistic phase, the lag may grow by no more than $T_p - 2T_r$.

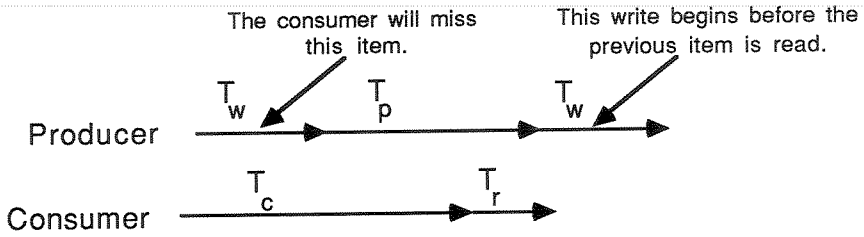

Figure 6a

It can be verified from figure 6b that the lag increases by the amount T_C + T_r - (T_W + T_p) during each produce-consume cycle. Therefore, during the final N-1 cycles, the lag grows by (N - 1)(T_C + T_r - (T_W + T_p).

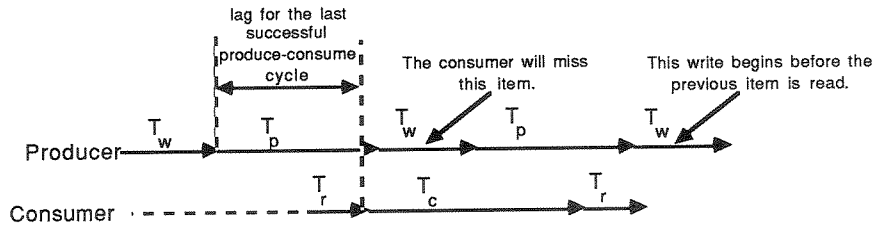

Figure 6b

To summarize, for the first produce-consume cycle, the lag between the completion of the producer's write to the buffer an the consumer's successful read from buffer is at most $2T_r$. For the optimistic phase to be error-free, the lag should not grow by more than T_p - $2T_r$ during the remaining N-1 produce-

consume cycles. The actual growth of the lag during those cycles is $(N - 1)(T_c + T_r - (T_w + T_p))$.

Therefore, the optimal level of optimism is obtained by choosing the largest N such that $(N - 1)(T_c + T_r - (T_w + T_p)) \leq T_p - 2T_r$; i.e.,

$$N \leq \frac{T_p - 2T_r}{T_c + T_r - T_p - T_w} + 1$$

Since the conjunct of conditions 1, 2 and 3 is identically true, it follows that these choices are necessary and sufficient to guarantee optimal performance.

Balanced Distributed Semaphores

Algorithms for message-based distributed systems can be, and often are derived from their shared variable counterparts. The semaphore was initially considered to be a primitive for shared variable systems. However, pessimistic "distributed" implementations of it have been proposed [SCH82]. This allows one to apply semaphore-based protocols to message-based distributed systems.

In distributed computing literature, the adjective "distributed" applied to "semaphore" [SCH82], "critical section" [MAE85, RIC81] or "lock" [THO79] has been used as a synonym for message-based implementations of these primitives in which all processes of the system participate equally; i.e., the role played by each process is identical. This description of "distributed" is not

quite the same as the description in Chapter 2, but is used in the following discussion of balanced distributed semaphores and locks for the purpose of maintaining consistency with past literature.

The three primitives, semaphores [DIJ68], critical sections [BRI73] and locks [BER87] are very similar. The **P** operation on a semaphore corresponds to the **prelude** to a critical section and the **lock** operation on a lock. The **V** operation on a semaphore, the **postlude** to a critical section and the **unlock** operation on a lock differ only as follows:

- An implementor may presume that the matching **prelude** and **postlude** to a critical section, and likewise, the matching **lock** and **unlock** operations on a lock are performed by the same process. There is no such presumption in the context of matching **P** and **V** operations on a semaphore.

- A semaphore or critical section does not have any property that corresponds to the "mode" of a lock. However, locks with modes can be simulated using multiple semaphores if and only if the following condition derived in [PAP81] is satisfied by the lock-modes:
  The availability of a lock in any given mode depends only upon the state of that lock.

  This is not a very restrictive condition. In fact, lock-modes defined by a compatibility matrix [BHG87] will always satisfy this condition.

Because of the similarities, adapting an implementation of one to implement either of the others is quite straight-forward. In this section, a pessimistic implementation of distributed critical sections is adapted to implement distributed semaphores. The resulting pessimistic implementation is then generalized to a balanced spectrum. Finally, the relationship between the characteristics of the distributed system and the optimal balance between optimism and pessimism is analytically derived. The next section presents a similar treatment of distributed locks.

Maekawa's distributed mutual exclusion algorithm

Summarized below is an efficient pessimistic implementation of distributed critical sections [MAE85] with a message overhead of $O(\sqrt{N})$ for a system of N processes.

With each process π_i, $0 \leq i < N$, is associated a subset of the processes $\mathbf{M}_i$. These subsets have the following properties:

- **The non-null intersection rule**:— It is required that every pair of these subsets have one or more common elements; i.e.,

  $\forall\, i, j: (0 \leq i, j < N) : M_i \cap M_j \neq \phi$.

- To distribute the overhead equitably among the processes, and incidentally, to make the algorithm more symmetric, it is desirable that

  - every process belong to about the same number of subsets and

- every subset contain about the same number of processes.

- Since the algorithm involves the exchange of messages between a process and the subset of processes associated with it, it is desirable that the sizes of the $\mathbf{M}_i$ be minimized.

- Since exchanging messages with a different process represents a greater overhead than exchanging messages with oneself, a slight improvement in the efficiency of the algorithm can be achieved if every process is a member of its $\mathbf{M}_i$.

For an optimal assignment of processes to the sets $\mathbf{M}_i$; i.e., one which minimizes the average number of processes in each set while satisfying the above requirements, each set contains approximately $\sqrt{N}$ processes and each process belongs to approximately $\sqrt{N}$ sets. The exact result, derived in [MAE85] is:

The number of processes required in each set to satisfy the above constraint for a system of N processes is the smallest integer k which satisfies the constraint $k^2 - k + 1 \geq N$.

Initially, the state of every process is "unlocked". A process requests the critical section by sending a "request" message to each element of its **M**. The receiver of a "request" message responds with a "grant" message if it is "unlocked." Otherwise, the request is queued. Upon granting a request, the

grantor becomes "locked." When a request is queued, appropriate deadlock-avoidance action is taken. The reader is referred to [MAE85] for details pertaining to avoidance of deadlock and starvation. The process enters the critical section after receiving a "grant" message from each element of its **M**. When the process leaves the critical section, it sends a "release" message to each element of its **M**. The receiver of a "release" message grants a queued request if any; otherwise it becomes "unlocked".

The Distributed Semaphore Algorithm

This balanced implementation of distributed critical sections can be adapted to implement distributed semaphores as follows:

A **P** operation is identical to the prelude to a critical section except that after receiving all the needed "grant" messages, the process sends a "commit" message to each element of its **M**. The receiver of a "commit" message becomes "committed" to the sender.

Definition 0:— A process π_p has **completed** a **P** operation if "grant" messages are received from all elements of $\mathbf{M}_p$.

If the process π_v performing a **V** operation is the same process π_p which completed the last **P** operation, it proceeds exactly as in the postlude to a critical section. Otherwise, the **V** operation is performed in two phases. In the first phase, π_p is identified. In the second phase, "release" messages are sent to the elements of $\mathbf{M}_p$.

complete a **P** operation, the elements of $M_1 \cap M_2$ should first become "unlocked". From the non-null intersection rule, at least one process should first become "unlocked". Since only a completed **V** operation can cause a process to become "unlocked", at least one **V** operation must be completed between two completed **P** operations.

QED

lemma 1

All "committed" processes are committed to the process which last completed a **P** operation.

Proof by induction:

Basis:— lemma 1 is true at all times up to the completion of the first **P** operation.

Proof of basis:— From lemma 0, no other **P** operation can be concurrently completed. A process can only become committed to one which has completed a **P** operation. Therefore, before the completion of the second **P** operation, any committed process can only be committed to the process which completed the first **P** operation.

Hypothesis:— lemma 1 is true at all times up to the completion of the i^{th} **P** operation.

Proof obligation:— lemma 1 is true at all times up to the completion of the $(i+1)^{th}$ **P** operation.

Proof:— From lemma 0, at least one **V** operation must intercede between the completion of the $(i-1)^{th}$ and the i^{th} **P** operation, say by processes π_{p0} and π_{p1} respectively. Consider the first such **V** operation to be completed by some process, say π_v.

1) From the description of the **P** operation, all elements of $\mathbf{M}_{p0}$ become committed to π_{p0} after the completion of the $(i-1)^{th}$ **P** operation.

2) From the induction hypothesis, a process can not be committed to any process other than π_{p0} between the completion of the $(i-1)^{th}$ and the i^{th} **P** operation. Further, since only an element of $\mathbf{M}_{p0}$ can be committed to π_{p0}, non-elements of $\mathbf{M}_{p0}$ are not committed to any process in this time-frame.

3) From the non-null rule applied to $\mathbf{M}_{p0}$ and $\mathbf{M}_v$, π_v will "identify" π_{p0} in phase 1 of the **V** operation.

When π_v completes its **V** operation, none of the processes are "committed". No process can become committed before the completion of the i^{th} **P** operation.

Between the completion of the i^{th} and the $(i+1)^{th}$ **P** operation, a process can only become committed as a result of the i^{th} **P** operation; i.e., a process can only become committed to π_{p1}.

QED

Theorem

The distributed semaphores algorithm is correct.

Proof obligation:— The completion of one **V** operation is necessary and sufficient between the completions of two successive **P** operations.

Proof:— Lemma 0 proves that the completion of one **V** operation is necessary between the completions of two successive **P** operations.

To prove sufficiency, let there be only one **V** operation, say by process π_v after the completion of a **P** operation by process π_p. The new proof obligation is that the **V** operation will be completed, and subsequently, all elements of $\mathbf{M}_p$; i.e., all processes "locked" by π_p will become "unlocked", thereby completely reversing the **P** operation.

Since π_p completes the **P** operation, all elements of $\mathbf{M}_p$ will eventually become "committed", and can not cease to be "committed" except as a consequence of the **V** operation. From the non-null intersection rule applied to $\mathbf{M}_p$ and $\mathbf{M}_v$ and lemma 1, π_v will "identify" π_p in phase 1 of the **V** operation,

and therefore, in phase 2 of the **V** operation, all elements of $\mathbf{M_p}$ will become "unlocked".

QED

Generalization to balanced protocols

A process optimistically passes a semaphore if the first **t** of **k** responses to its "request" message are "grant" messages, where **k** is the number of elements in its set **M** (defined earlier) and **t** is a number between zero and **k** which determines the level of optimism at which the protocol operates. If any of the first **t** responses is a "blocked" message, the process behaves pessimistically. Thus, the parameter **t** defines the level of optimism. A process which optimistically passes a semaphore "detects a false start" if any of the **k - t** remaining responses to its "request" message are "blocked" messages. To compensate for its incorrect action, the process "rolls back" to its state before passing the semaphore and waits for all remaining requests to be granted. If, during the interval between the optimistic passing of the semaphore and the detection of the false start, messages were sent to other processes, thereby possibly compromising their states, they must be rolled back to their state prior to receiving those messages. These processes may in turn cause others to whom they sent messages to roll back. Techniques for coordinating the roll-back of a system of processes to a previous consistent state are discussed in [JEF85, WOO80, RUS80, STR85, KOO87, JOH88]

Analysis: choosing the best balance

Consider a distributed semaphore in a distributed system of **N** processes. Quantities **k** and **t** used in this analysis are defined in the previous paragraph. Let **r** be the average rate at which each process performs **P** operations on the semaphore and μ be the average length of the period during which a request blocks the semaphore at that process. "Request" messages are generated at the average rate of $\mathbf{k \cdot r \cdot N}$ and each process receives "request" messages at an average rate of $\mathbf{k \cdot r}$. Therefore, the fraction of time during which a semaphore is blocked at a process is $\mu \cdot \mathbf{k \cdot r}$. For brevity, let **B** represent this quantity. Assuming that **P** operations on the semaphore are independent random events, the probability that a semaphore is blocked at a process at the instant a "request" message is received is **B**. Therefore, the response to a "request" message will be a "blocked" message with probability **B** and a "grant" message with probability (1 - **B**).

For a process to optimistically pass a semaphore, the first **t** responses to its "request" messages must be "grant" messages. This optimistic decision will turn out to be correct if all remaining **k** - **t** responses are also "grant" messages and wrong if at least one of the remaining **k** - **t** responses is a "blocked" message. The probability that the first **t** responses are "grant" messages is $(1-\mathbf{B})^{\mathbf{t}}$. The probability that all the remaining **k** - **t** responses are also "grant" messages is $(1-\mathbf{B})^{(\mathbf{k}-\mathbf{t})}$. The probability that at least one of these **k** - **t** responses is a "blocked" messages is $(1- (1-\mathbf{B})^{(\mathbf{k}-\mathbf{t})})$. Therefore, the probability that the process optimistically and correctly passes a semaphore is

$(1-\mathbf{B})^{t}(1-\mathbf{B})^{(k-t)} = (1-\mathbf{B})^{k}$. The probability that the process wrongly passes a semaphore is $(1-\mathbf{B})^{t}(1-(1-\mathbf{B})^{(k-t)}) = (1-\mathbf{B})^{t} - (1-\mathbf{B})^{k}$.

Let $\mathbf{E(t)}$ represent the average time a process must wait to receive $\mathbf{t}$ of the responses to its "request" messages. Every time a correct optimistic decision is made, the average time saved is $(\mathbf{E(k)} - \mathbf{E(t)})$. Let the expected time lost to a wrong optimistic decision be $\mathbf{W}$. The expected time gained per $\mathbf{P}$ operation, T, is (the probability of a correct optimistic decision times the average time saved when a correct optimistic decision is made) minus (the probability of a wrong optimistic decision times the average time lost to a wrong optimistic decision); i.e.,

$$T = (1-\mathbf{B})^{k}(\mathbf{E(k)} - \mathbf{E(t)}) - ((1-\mathbf{B})^{t} - (1-\mathbf{B})^{k})\mathbf{W} \quad \text{.....(1)}$$

or, since $\mathbf{B}$ is an abbreviation for $\mu\cdot\mathbf{k}\cdot\mathbf{r}$,

$$T = (1-\mu\cdot\mathbf{k}\cdot\mathbf{r})^{k}(\mathbf{E(k)} - \mathbf{E(t)}) - ((1-\mu\cdot\mathbf{k}\cdot\mathbf{r})^{t} - (1-\mu\cdot\mathbf{k}\cdot\mathbf{r})^{k})\mathbf{W} \quad \text{.....(2)}$$

The optimum balanced protocol corresponds to the choice of $\mathbf{t}$ from the range 0 to $\mathbf{k}$ which maximizes T.

Example

As an example, consider the case where a process needs to "request" 9 other processes to pass a semaphore. This corresponds to an implementation of a distributed semaphore on a system of 91 processes. Round-trip message delays are assumed to be exponentially distributed. All time units are

normalized with respect to an average message delay. The range of values of recovery costs used in this and a later example are chosen to best highlight the transition of the optimal from optimistic through balanced to pessimistic sequencing. Data on recovery costs encountered in real-world systems was not obtainable despite our best effort.

Assuming a round-trip message delay is exponentially distributed with mean **m**, the expected time for the arrival of **t** of **k** responses is given by the expression:

$$\mathbf{m}\sum_{\mathbf{i=0}}^{\mathbf{t-1}}\frac{1}{\mathbf{k-i}} \qquad \text{.....(3)}$$

This expression was derived using Mathematica [WOL88], a symbolic mathematics software package.

The analysis of this example is summarized in figures 7 and 8. The optimal balance and the improvement it offers over the pessimistic protocol were computed for a range of operating conditions.

Figure 7 is a tabular representation of the results. Its columns from left to right represent increasing values of the expected cost (in terms of message delays) of recovering from an incorrect **P** operation. Its rows from top to bottom represent decreasing availability of the semaphore, where availability is the fraction of time a semaphore (i.e., not just one, but all the replicas to which

requests are to be sent) is unblocked, and equals $(1-\mu \cdot \mathbf{k} \cdot \mathbf{r})^{\mathbf{k}}$. The two numbers in each cell represent the optimal balance, **t** and the average time gained per **P** operation relative to the pessimistic protocol, T.

Rec. Costs	0.16	0.32	0.64	1.28	2.56	5.12	10.24	20.48
Availability								
1	0 5.66e+00	0 5.66e+00	0 5.66e+00	0 5.66e+00	0 5.66e+00	0 5.66e+00	0 5.66e+00	0 5.66e+00
0.95	0 5.37e+00	0 5.36e+00	0 5.34e+00	0 5.31e+00	0 5.25e+00	0 5.12e+00	0 4.86e+00	0 4.35e+00
0.9	0 5.08e+00	0 5.06e+00	0 5.03e+00	0 4.96e+00	0 4.84e+00	0 4.58e+00	0 4.07e+00	2 3.09e+00
0.85	0 4.79e+00	0 4.76e+00	0 4.71e+00	0 4.62e+00	0 4.43e+00	0 4.04e+00	0 3.27e+00	5 2.24e+00
0.8	0 4.49e+00	0 4.46e+00	0 4.40e+00	0 4.27e+00	0 4.01e+00	0 3.50e+00	3 2.61e+00	6 1.67e+00
0.75	0 4.20e+00	0 4.16e+00	0 4.08e+00	0 3.92e+00	0 3.60e+00	0 2.96e+00	4 2.09e+00	7 1.24e+00
0.7	0 3.91e+00	0 3.86e+00	0 3.77e+00	0 3.58e+00	0 3.19e+00	2 2.48e+00	5 1.69e+00	7 9.17e-01
0.65	0 3.62e+00	0 3.57e+00	0 3.45e+00	0 3.23e+00	0 2.78e+00	3 2.08e+00	6 1.36e+00	8 6.47e-01
0.6	0 3.33e+00	0 3.27e+00	0 3.14e+00	0 2.88e+00	1 2.38e+00	4 1.73e+00	7 1.06e+00	8 4.82e-01
0.55	0 3.04e+00	0 2.97e+00	0 2.82e+00	0 2.54e+00	2 2.02e+00	5 1.43e+00	7 8.50e-01	8 3.26e-01
0.5	0 2.75e+00	0 2.67e+00	0 2.51e+00	0 2.19e+00	3 1.70e+00	6 1.17e+00	7 6.47e-01	8 1.80e-01
0.45	0 2.46e+00	0 2.37e+00	0 2.19e+00	1 1.85e+00	4 1.41e+00	6 9.47e-01	8 4.72e-01	8 4.50e-02
0.4	0 2.17e+00	0 2.07e+00	0 1.88e+00	2 1.54e+00	5 1.15e+00	7 7.37e-01	8 3.61e-01	9 0.00e+00
0.35	0 1.88e+00	0 1.77e+00	0 1.56e+00	3 1.26e+00	5 9.26e-01	7 5.79e-01	8 2.57e-01	9 0.00e+00
0.3	0 1.59e+00	0 1.47e+00	1 1.26e+00	4 1.00e+00	6 7.21e-01	7 4.29e-01	8 1.60e-01	9 0.00e+00
0.25	0 1.29e+00	0 1.17e+00	2 9.86e-01	4 7.70e-01	6 5.41e-01	7 2.88e-01	8 7.37e-02	9 0.00e+00
0.2	0 1.00e+00	1 8.84e-01	3 7.34e-01	5 5.66e-01	7 3.80e-01	8 1.99e-01	9 0.00e+00	9 0.00e+00
0.15	0 7.13e-01	2 6.16e-01	4 5.06e-01	6 3.81e-01	7 2.49e-01	8 1.20e-01	9 0.00e+00	9 0.00e+00
0.1	2 4.39e-01	4 3.74e-01	5 3.03e-01	6 2.19e-01	7 1.29e-01	8 5.07e-02	9 0.00e+00	9 0.00e+00
0.05	4 1.94e-01	5 1.64e-01	6 1.28e-01	7 8.95e-02	8 4.94e-02	9 0.00e+00	9 0.00e+00	9 0.00e+00
0	9 0.00e+00	9 0.00e+00	9 0.00e+00	9 0.00e+00	9 0.00e+00	9 0.00e+00	9 0.00e+00	9 0.00e+00

Figure 7

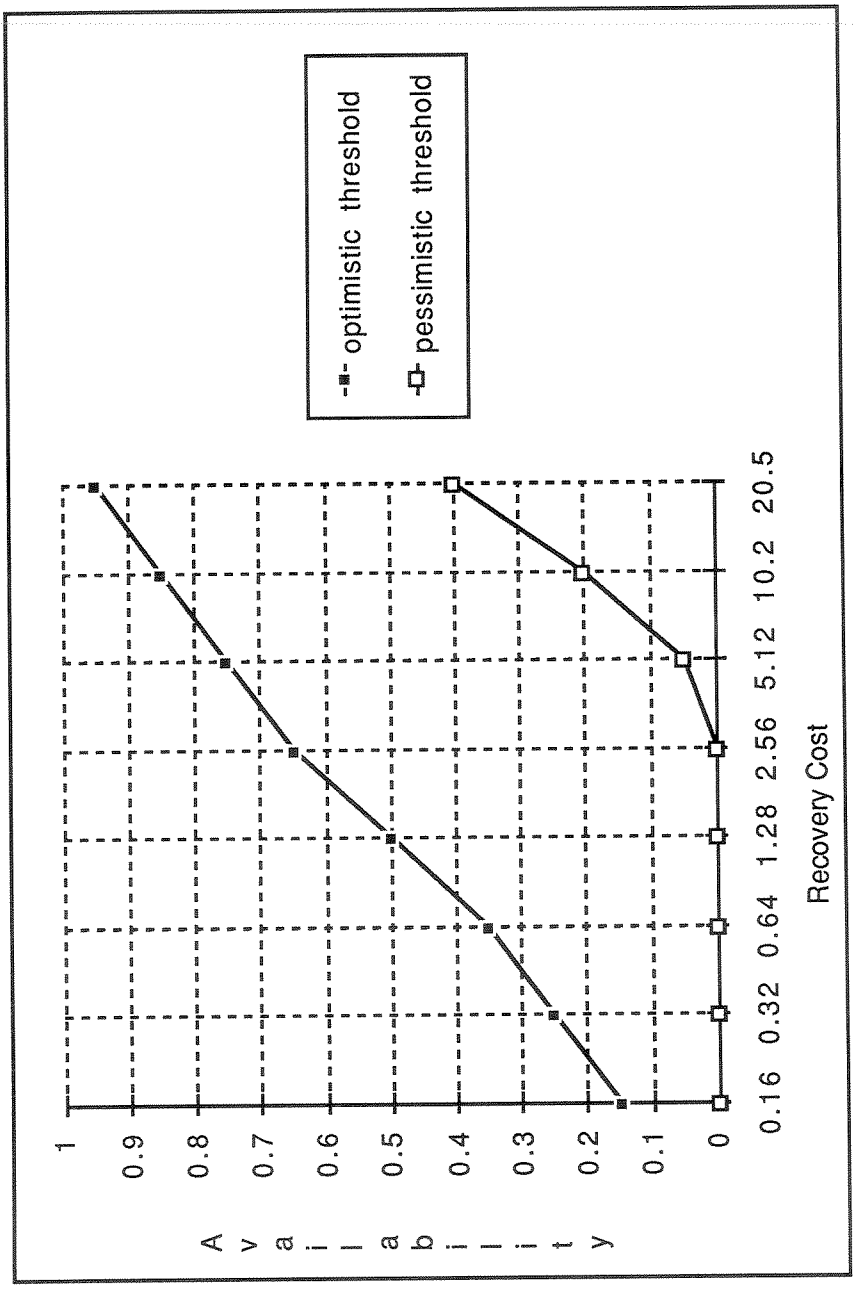
optimistic threshold
pessimistic threshold
Availability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.16 0.32 0.64 1.28 2.56 5.12 10.2 20.5
Recovery Cost

Figure 8

The graph of figure 8 is derived from the tabulated data in figure 7. The X and Y axes of the graph correspond to the columns and rows of the table. Extreme optimism is optimal in the region above the optimistic threshold and extreme pessimism is optimal in the region below the pessimistic threshold. It is in the region between these lines that balanced protocols are optimal.

The following conclusions can be drawn from the results obtained for this example:

- Higher availability of the semaphore and lower recovery costs tend to favor more optimistic protocols. Likewise, lower availability of the semaphore and higher recovery costs tend to favor more pessimistic protocols.

- For this example, the introduction of the optimal level of optimism typically saves a few message delays— about 0.5 to 5— per **P** operation. This suggests that the introduction of optimism can only be effective in situations where message delays have a significant impact on performance.

Balanced Distributed Locking

The lock, a popular synchronization primitive for database concurrency, can be viewed as a simple adaptation of the semaphore as described in the previous section. **Lock** and **Unlock** operations on a Lock data-type correspond to the **P** and **V** operations on a Semaphore data-type. The adaptation consists of a

generalization, the introduction of "lock modes", and permits a specialization by allowing the implementor to assume that every **Lock** operation will be matched by an **Unlock** operation from the same process.

The primary distinction between the Lock data-type and the semaphore data-type is the notion of lock modes. The reader may recall that the state of a semaphore consists of a queue of blocked requests and a binary variable of type {unblocked, blocked}. For a lock, instead of the binary variable, there is a set-valued variable. The elements of this set are pairs <M, P> where M is a lock mode and P is a process (or transaction) which holds the lock in mode M.

Several sophisticated locking protocols require an additional class of operations on a lock called **lock conversions**. The result of a successful **lock conversion** is that the holder of a lock in some initial mode prior to that operation ends up holding the lock in a different final mode. Lock conversion is discussed in detail in [BER87]. A hurdle to adapting an implementation of **balanced replicated semaphores** to implement **balanced replicated locking** is that there is no simple relationship between a **lock conversion** operation and the standard **P**, **V** operations on a semaphore. To overcome this hurdle, a simple and reasonable alternative to classical lock conversion is proposed below. In the interest of simplicity, this discussion is limited to protocols which obey the two-phase rule.

Classical Two-phase locking with lock conversion

The following are the classical rules for two-phase locking [ESW76] and lock conversion [GRA75].

Rule 2PL:— A transaction shall not acquire any lock after it has released a lock.

Rule CLC:— A transaction wishing to convert a lock it holds in mode $\mathbf{M_0}$ to mode $\mathbf{M_1}$ shall instead convert it to some mode $\mathbf{M_3}$ such that $\mathbf{M_3}$ is incompatible with any mode which is incompatible with either $\mathbf{M_0}$ or $\mathbf{M_1}$. An example of the application of this rule is in [BER87], section 3.9.

Such an $\mathbf{M_3}$ is said to be **stronger** than both $\mathbf{M_0}$ and $\mathbf{M_1}$. For efficiency, the weakest $\mathbf{M_3}$ which conforms to **Rule CLC** must be chosen. A correct protocol which imposes **Rule 2PL** remains correct if it allows lock conversions according to **Rule CLC** during the first (locking) phase.

A formal theory of lock modes and the following generalization of two-phase locking are presented in [KOR83].

A Generalization of two-phase locking

The combination of **rules 2PL** and **CLC** is actually stronger than necessary. They may be combined and weakened to the following:

Rule G2PL:— A transaction shall neither acquire nor strengthen a lock after it has either released or weakened a lock.

An alternative to lock conversion

Instead of allowing lock conversion, it suffices to allow a transaction to hold multiple locks on a datum, and note that [BER87] no two locks held by the same transaction can conflict. The equivalents to strengthening and weakening a lock is acquiring additional locks in different modes on the same datum and releasing some of the locks held on the datum respectively. With this simplification, **rule 2PL** is all that is needed. Because the need for lock conversion is eliminated, adapting the distributed semaphore algorithm to implement distributed locking becomes straightforward.

If the protocol designer is pre-assigned a set of lock-modes and a compatibility matrix which may not altered, this scheme admits more schedules than **Rule G2PL** because the weakest available mode to which a lock can be converted may be stronger than the combination of lock modes required by a transaction.

Adapting the implementation of Balanced Semaphores

Distributed semaphores may be adapted to implement distributed locking as follows:

The Lock operation

The "request" message specifies the mode in which the requestor wishes to hold the lock. A request is granted (by responding with a "grant" message) when no other transaction holds the lock at that process in a conflicting mode. A consequence of the alternative to lock-conversion described earlier is that the

transaction which originated the request may already be holding the lock in a conflicting mode. This situation does not prevent the request from being granted. A transaction is allowed to hold a lock in an arbitrary number of conflicting modes at the same time. In other words, two requests from the same transaction will never conflict, even if their modes conflict. There is also no need for "commit" messages.

The Unlock operation

Phase 1 of the **V** operation is not required, since the initiator of the **unlock** is also the initiator of the matching **lock**. A "release" message specifies which of the modes currently held on a lock should be released. Blocked requests if any, which do not conflict with the remaining granted requests are granted.

Adapting the analysis of Balanced Semaphores

The analysis leading to the determination of the optimal balance for a replicated semaphore is also valid for a replicated lock with the following adaptation:

- In the place of the parameter **A**, which denoted the availability of a semaphore, use one of a family of parameters A^{i}_{m}, where **i** denotes the item being locked, **m** denotes the mode in which the lock is being requested, and A^{i}_{m} denotes the availability of item **i** in mode **m**.

- Assuming a simple model of transaction behavior, a more realistic model of recovery costs can be constructed. In the case of the semaphore, recovery-cost was modelled as a random variable with a time-invariant mean. Database systems typically employ **abort-and-restart** as a recovery

strategy. Also, the extent of modifications to the database, and hence, the cost of undoing these modifications when required by an abort, increases with the progress of a transaction. To reflect this fact, the analysis of the semaphore is refined as follows:

- In the place of the parameter **W**, which denotes the expected time-cost of a wrong optimistic decision, use one of a family of parameters, $\mathbf{W_p}$, where **p** is the number of pages that will have to be restored, and $\mathbf{W_p}$ is the time-cost of recovery if **p** pages have to be restored. Further, if $\mathbf{W_p}$ is proportional to **p**, we have $\mathbf{W_p = W_1 \cdot p}$.

With these modifications to the analysis, the expected time gained per lock operation is given by the equation:

$$T = (A^i_m)^k(E(k) - E(t)) - ((A^i_m)^t - (A^i_m)^k)W_1 \cdot p \quad \text{.....(4)}$$

Finally, as with semaphores, the optimum balanced protocol corresponds to the choice of **t** from the range 0 to **k** which maximizes the above expression.

Example

To demonstrate the application of the above analysis, the optimal balance and the performance improvement it achieves is determined for a system previously evaluated in a simulation study of distributed databases by Carey and Livny [CAR88].

The system simulated in this study has the following characteristics:

- number of sites:— 8
- size of the database:— 8 groups of 3 files each. Each file has 800 pages. Locks are obtained at the page-level.
- work-load:— 50 transactions per site. There are 50 terminals per site, each executing a transaction. Upon termination of a transaction, another one is immediately initiated at that terminal.
- transaction profile:— 18 pages are locked by each transaction, all of which belong to the same group of files. Each lock obtained is a write-lock with probability 1/4 and a read lock with probability 3/4.

These are the characteristics relevant to this example. In addition to these, the original study also models the resources at each site; one CPU and two disks. The variables of the study include the think-time between lock requests, the CPU time required to send or receive a message, locality of data and the degree of replication. The simulation varied these values to achieve various levels of resource contention, and compare the throughput achieved using a wide range of concurrency management protocols. In fact, the effects of resource contention was the focus of the original study. Nevertheless, their parameters were adopted for this example, since they represent a reasonable model of a distributed database and its workload.

In the interest of simplicity and to keep the example focused on balanced sequencing, the resource-contention aspects of the study are ignored in this analysis. The database is assumed to be fully replicated, and it is assumed that there are enough resources to eliminate resource contention. The issue of message transition delay is ignored in the simulation study, but, as can be seen from equation (4), it is important in the context of this problem. It is therefore assumed that, as in the distributed semaphore example, the round-trip message delay is exponentially distributed.

Since all the pages accessed by a transaction belong to the same group of files, in the absence of resource contention, there is no interaction between transactions accessing different groups of files. Therefore, one need only study transactions accessing one of the eight groups of files. Further, the notion of a file as an element of a group can be dropped, and a group can be considered to consist of 2400 pages. So, an equivalent system would be:

- number of sites:— 8
- size of the database:— 2400 pages.
- work-load:— 50 transactions (50 transactions per site * 8 sites / 8 groups of files)

- transaction profile:— 18 pages are locked by each transaction. Each lock obtained is a write-lock with probability 1/4 and a read lock with probability 3/4.

Assuming that the number of locks held by a transaction progresses linearly with time, the average number of locks held by each of the 50 transactions will be 9. (18/2) Thus, the average number of locks held by all 50 transactions will be 450. (9*50) Of these, 112.5 (450/4) will be write locks and 387.5 (450*3/4) will be write locks. Given that read-locks are shared and write-locks are exclusive, the availability of a page is determined to be approximately 0.82 for write-locks and 0.95 for read-locks. At the pessimistic end-point, a lock should be acquired at four sites to lock a database entity.

One parameter used in the derivation of optimal balance can not be determined from the data used in the Carey-Livny study, and that is the cost of recovering from an incorrect optimistic decision. For the purpose of this example, this value was varied from 0 to 20 time units, since this range of values best demonstrates the transition of the optimal protocol from optimistic to pessimistic. As mentioned earlier, all time units are normalized with respect to the average message delay. The optimal protocol for each of the lock-modes and the average reduction achieved in transaction run-time is tabulated as a function of the cost of recovery in figure 9.

Recovery Cost	OBWL	OBRL	RTRT
0	0	0	54.3
1	0	0	49.3
2	1	0	44.9
3	2	0	41.4
4	3	0	38.1
5	3	0	35.1
6	3	0	32.2
7	3	0	29.2
8	3	0	26.3
9	3	1	23.9
10	4	1	22
11	4	1	20.2
12	4	2	18.7
13	4	2	17.5
14	4	2	16.4
15	4	2	15.2
16	4	2	14
17	4	2	12.8
18	4	2	11.6
19	4	3	11
20	4	3	10.4

OBWL = Optimal Balance for Write Lock
OBRL = Optimal Balance for Read Lock
RTRT = Reduction in Transaction Run Time

Figure 9

For this example, balanced locking is optimal for write-locks if the average recovery cost is 9 message delays or less. For read-locks which exhibit greater availability, (95% vs. 82% for write-locks) optimistic or balanced locking remains optimal for recovery costs as high as 20 message delays. Unfortunately, hard data on typical recovery costs could not be obtained despite our best effort. The data shows that reduction in transaction run-time is quite sensitive to increase in recovery cost.

Chapter 5
The Echo Phenomenon

In this section, a phenomenon called the "echo" is identified whereby a single timing fault under an optimistic protocol can result in extreme and potentially permanent degradation of a system's performance. Even under more benign conditions, where such a permanent degradation does not arise, it is demonstrated that this phenomenon may exact a significant performance cost. The manifestation of the echo phenomenon is illustrated with a case study using a time warp [JEF85] protocol. Finally, a technique for eradicating echoes is proposed.

The premise: why optimism is expected to work

A necessary criterion for the employment of optimism to enhance performance is that faults occur with sufficiently low probability that performance degradation resulting from wrong optimistic decisions does not offset performance gained because other such decisions happened to be right. To justify the incorporation of optimism in a protocol, it is typically established that given the timing characteristics of the processes and a correct global state, the probability that each subsequent optimistic decision results in a timing fault is some sufficiently low value. The expected performance is then estimated based on this low probability of a timing fault.

The pitfall: why optimism may not work

The fallacy in this reasoning is the assumption that the timing relationship between processes are not disturbed by the occurrence of a timing fault. In fact, this may not be the case, since some of a process' routine activity will be suspended while recovery is in progress. This disturbance can increase the probability of a timing fault elsewhere in the system, resulting in additional timing faults, and hence, additional disturbances of the timing characteristics. In the worst case, these "echoes" of timing faults could be self-perpetuating, resulting in an average rate of progress which asymptotically approaches zero. Even if the echoes do not self-perpetuate, they could lead to significantly poorer performance. However, as proven in [JEF85], a system employing a time warp protocol will continue to make some progress regardless of the rate at which faults occur.

Example: the Time Warp protocol

This section illustrates the echo phenomenon using a network of processes sequenced using the Time Warp [JEF85] protocol. A typical application for this protocol is distributed simulation. This problem has been chosen because:

- it is currently of interest to a large body of researchers
- optimistic [JEF85] and pessimistic [CHA81] protocols have been proposed and

- proposed protocols are relatively easy to analyze partly because of their operational simplicity.

This study will demonstrate that for cyclic networks of processes, the echo phenomenon can result in serious performance degradation. For systems consisting of simple cycles, the degradation is quantified—analytically for a cycle of two nodes and numerically for larger cycles.

virtual time and time-warp

The concept of virtual time and the time-warp protocol are described in [JEF85] and summarized in chapter 6. An understanding of this material is required for the rest of this chapter.

The echo in a ring

Consider the two-process system of figure 10. If the time-warp protocol is to perform well, large discrepancies between the processes' virtual times should be unlikely. To isolate the effect of the echo, it is therefore assumed that the processes progress at the same rate. Under that assumption, one is inclined to expect that large timing faults; i.e., large discrepancies between a process' virtual time and the time-stamp of a message it receives, are infrequent. The following analysis will show that a single small timing fault can result in a self-perpetuating series of progressively larger faults.

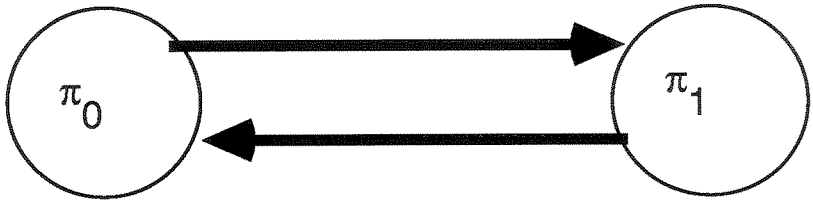

figure 10

Consider the receipt of a message time-stamped T from π_1 by π_0 when its virtual time is T+∂, resulting in a timing fault of size ∂. Suppose that while π_0 rolls back to T, π_1 advances to T+∂. Because the processes progress at the same rate, this discrepancy will persist until π_1 receives a message from π_0. At that time, π_1 will experience a timing fault of size ∂. Thus, the discrepancy between the virtual times of π_0 and π_1 will oscillate, causing a perpetual series of timing faults of size ∂.

However, if roll-back were faster than progress, the above oscillation will be damped. Suppose that while one process rolls back by ∂, the other can only advance by some $\mathbf{f}(\partial) < \partial$. Then, each echo will reduce the discrepancy, thereby damping the echo. Likewise, if $\mathbf{f}(\partial) > \partial$, the oscillation will build up unboundedly. For more complex networks, the necessary and sufficient condition on **f** that ensures damping of echoes could be more restrictive and harder to determine. At any rate, even a gradually damped echo can exact a

high performance cost from a timing fault. A later section describes a protocol which ensures the efficient eradication of echoes regardless of **f**.

Quantifying echoes

The analysis of the echo comprises the following steps:

1) Determine the effect of a message exchanged between processes as a function of their progress and recovery rates, the initial discrepancy in their virtual times and the message delay.

2) Using 1), quantify the consequences of the echo phenomenon for a ring of two processes.

3) Generalize 2) using numerical iteration over 1) to analyze an arbitrary-sized ring of processes.

the effect of one message

Consider two processes, π_0 and π_1, in an arbitrary network. Figure 11 depicts their progress in virtual time as a function of real time. Let **P** be the nominal rate at which a process progresses; i.e., the rate at which it would progress if it never had to recover. Let **R** be the rate at which a process recovers. **m** denotes a message delay, the elapsed time between the sending of a message and its receipt. ∂ denotes an initial discrepancy between the virtual times of the two processes.

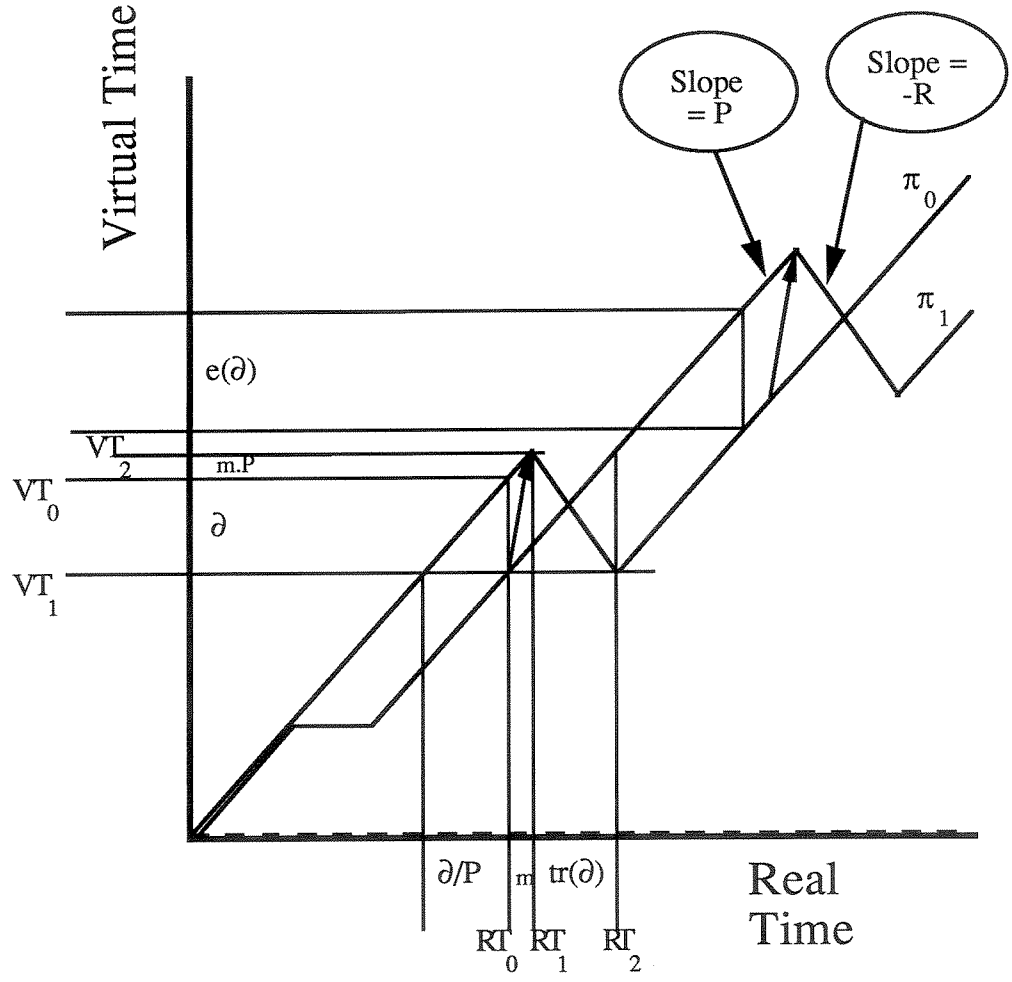

figure 11

The sequence of events depicted in figure 11 are as follows:— At real time RT_0, the virtual times are VT_0 at π_0 and VT_1 at π_1, with both processes

progressing at a rate of **P** virtual time units per real time unit and π_1 is ∂ behind π_0 in virtual time. At RT_0, π_1 sends a message time-stamped VT_1 which is received by π_0 **m** real time units later at RT_1 when the virtual time at π_0 is VT_2. This causes π_0 to roll back to virtual time VT_1, the time-stamp on the message, at a rate of **R** virtual time units per real time unit. The roll back is completed at real time RT_2 and π_0 resumes progressing at rate **P**. The time spent rolling back, **tr**(∂) and the final discrepancy in virtual times, **e**(∂), the "echo" of a timing fault of size ∂, can be computed using the laws of geometry and the facts that the positive-sloped lines in figure 11 have a slope of **P** and the negative-sloped lines have a slope **-R**. They are:

$$tr(\partial) = \frac{\partial + m.P}{R} \quad \text{...(5)}$$

and

$$e(\partial) = \frac{P}{R}(\partial + m(P + R)) \quad \text{..(6)}$$

analysis of a 2-process ring

Consider the ring of two processes, π_0 and π_1, (figure 10) synchronized by time warp. Suppose that initially, π_0 is ∂ behind π_1 in virtual time of the two processes. As described above, a message from π_0 to π_1 results in π_0 spending time **tr**(∂) recovering and π_1 being **e**(∂) behind π_0. Subsequently, a message from π_1 to π_0 results in π_1 spending time **tr**(**e**(∂)) recovering and π_0 being **e**(**e**(∂)) behind π_1. Thus, the $\mathbf{n^{th}}$ such message will result in time $\mathbf{tr(e^{n-1}}(\partial))$ being spent on recovery.

From equations 5 and 6:

$$tr(e^{n-1}(\partial)) = \frac{\frac{P}{R}(\frac{P}{R}(\frac{P}{R}\ldots n\text{-}1 \text{ times}(\partial + m(P + R)) + m(P + R)) + \ldots n\text{-}1 \text{ times}) + m.P}{R}$$

After substituting X for P/R and Y for m(P+R):

$$= \frac{X(X(X\ldots n\text{-}1 \text{ times}(\partial + Y) + Y) + \ldots n\text{-}1 \text{ times}) + m.P}{R}$$

$$= \frac{\partial.X^{n-1} + Y(X^{n-1} + X^{n-2} + \ldots + X) + m.P}{R}$$

$$= \frac{\partial.X^{n-1} + Y(\frac{X^{n} - X}{X - 1}) + m.P}{R} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(7)$$

The following observations may be made of equation (7):

- The contributions to the performance degradation of the initial discrepancy in virtual time, ∂, and the message delay, **m**, are mutually additive, and hence, can be studied independently.
- If X < 1, in the limit n—>∞, the contribution of ∂ approaches 0 and the contribution of **m** approaches a constant.
- If X > 1, the contributions of both terms are unbounded.

analysis of larger rings

The progression of echoes in rings of more than two processes was numerically evaluated using equations 5 and 6. The asymptotic limit of the

echo as time progresses to infinity was determined by monitoring the normalized first- and second- differences of its progression. Starting with a ring where all processes have the same virtual time except for one which lags behind by ∂ time units, the asymptotic limits of the echo were determined as follows:

Let ∂_i represent the size of the echo after it progresses **i** times around the ring and **e** an arbitrarily small value. The echo is determined to have ceased to grow when

$$\frac{\partial_i - \partial_{i-1}}{\partial_{i-1}} < \varepsilon$$

i.e., the normalized first difference is small and

$$\partial_i - \partial_{i-1} < \partial_{i-1} - \partial_{i-2}$$

i.e., the second difference is negative.

$\partial_i - \partial_{i-1} > \partial_{i-1} - \partial_{i-2}$ is evidence that the growth of the echo is unbounded. To avoid drawing conclusions from possibly transient behavior, data about the first several propagations of the echo around the ring were ignored.

The following conclusions were drawn from the preceding study:

- For a ring of N processes, the progression of the echo is bounded iff $R > (N - 1)P$.

- Where the previous condition is satisfied, the echo asymptotically grows to its limit of:

$$\frac{m.\frac{P}{R}(R + 1)(N - 1)}{1 - (N - 1)\frac{P}{R}}$$

It is reassuring to observe that these results are consistent with the analytical results obtained for the case where $N = 2$.

A solution: an echo-damping protocol (edp)

This solution was inspired by Dijkstra's paradigm of diffusing computations [DIJ80].

Outline

When a process senses an echo, it initiates a diffusing computation which causes all processes to switch to their pessimistic protocol.

With all processes employing their pessimistic protocol, no additional timing faults will occur. Eventually, all timing faults will be recovered from, and the network will reach a stable state of freedom from timing faults which is detected by the diffusing computation.

When this stable state is detected, processes resume optimistic operation.

Details

A process which senses an echo and is not already a participant in **edp** initiates **edp** by appointing itself the root of an echo damping tree (**edt**). The **edt** is assigned a unique identity. The protocol defines a total order, Ω of **edt**s based on their identities. The choice of the unique identities and the total order is discussed later in this section.

The process then sends an echo damping message (**edm**) containing the identity of the **edt** to each of its immediate neighbors and switches to its pessimistic protocol

The recipient of an edm processes it as follows:

- If the recipient already belongs to a "better" **edt** in the total ordering Ω, it ignores the message.
- If the recipient already belongs to the same **edt**, it "accounts for" the sender, but otherwise ignores the message.
- Otherwise, the recipient joins the **edt**, appoints the sender as its parent, switches to its pessimistic protocol, "accounts for" the sender and sends the **edm** to each of its immediate neighbors **except its parent**.

A process which "accounts for" all its immediate neighbors can not have any children in the **edt**, since a child never sends an **edm** to its parent. It recovers from any timing faults it may have experienced, sends an **edm** to its

parent and removes itself from the **edt**. However, it continues to be pessimistic unless it is the root of the **edt**.

When the root of the **edt** removes itself from the **edt**, it must be the case that the network is free of timing faults. It initiates the resumption of optimistic operation. The resumption of optimistic operation is straight-forward. The initiator sends resume-optimism (**ru**) messages to all its immediate neighbors and switches to its optimistic protocol. A recipient of **ru** who is already optimistic ignores the message. Otherwise, the recipient sends **ru** to all its immediate neighbors except the sender and switches to its optimistic protocol.

It remains to choose the total ordering Ω. The choice is an issue of performance and does not affect correctness. Ideally, one would like the **edt** which has spread the farthest to take precedence. This is likely to be the older **edt**. This suggests identifying an **edt** with its creation time; i.e., the local time of its root when it was created, and the identity of its root, to be used to resolve identical creation times. The lower the creation time, the better the edt. Where creation times are equal, the id of the root is used to determine the order in Ω.

The proof of correctness of this algorithm is similar to the proof of correctness of the termination detection algorithm in [DIJ80] which employs the same paradigm.

Chapter 6
Survey of Related Literature

This research rests heavily on a large body of literature spanning several domains. The purpose of this chapter is to relate this research to the work of others.

This chapter is organized in two sections. The "concepts survey" section discusses the relevance of some concepts from literature to this research. The "literature survey" section summarizes the contribution of prominent articles to the development and establishment of the concepts discussed in the "concepts survey".

Concepts Survey

[KUC81] discusses a categorization of dependences among steps of a program. Dependences in a program are treated as implicit properties of its semantics and the paper deals with determining dependences at compile time for the purpose of realizing the potential for concurrency in a sequentially formulated program. In contrast, dependences are explicitly (and syntactically) specified in our program notation. Further, in contrast to [KUC81]'s five classes of dependences, we need to distinguish between only three classes of dependences.

Protocols for preserving dependences among steps of a program employ some form of sequencing primitive. Primitives proposed for this purpose include "atomic memory fetch", "test and set", time-stamps, event-counts and sequence numbers [REE79], semaphores [DIJ68] and extended semaphores [AGE77], locks [ESW76] and broadcasts [SCH82]. Of these, semaphores and the closely related locks are the most widely used, and balanced implementations of these are presented in chapter 4.

The paradigms underlying the implementation of distributed mutual exclusion in [RIC81], distributed locking in [THO79] and distributed mutual exclusion in [MAE85] represent a hierarchy of increasingly efficient approaches to distributed decision-making. [RIC81] involves every process in every decision, [THO79] involves at least half the processes and [MAE85] involves about $\sqrt{N}$ of N processes.

Balanced protocols employ logging and checkpointing to support the detection of inconsistencies arising from the optimism in the triggering of events and the subsequent recovery from those inconsistencies. This use of the log bears a strong semblance to the use of logs in the Gypsy environment [GOO1, 2, 3] for run-time verification of programs. The goals are different but the approaches are comparable.

State restoration and recovery have been extensively studied in the context of crash tolerance, but the techniques and some of the results on performance

are relevant in the context of recovering from computational inconsistencies. [WOO80, STR85, KOO87, JOH88] discuss implementation techniques, [RUS80] treats the problem of avoiding cascading rollback and [CHA75, CHA81, GEL76] derive analytical results for performance.

[AGR85] discusses the common overhead of consistency management and crash recovery mechanisms and presents a case for integrating them. Balanced sequencing is conducive to such integration since the logging and recovery mechanisms serve both needs. Consequently, where crash recovery is a requirement, the overhead associated with the implementation and execution of the logging mechanism and the implementation of the recovery mechanism need no longer be attributed to the potential for optimism in this approach to consistency management, thus strengthening the case for permitting optimism.

Literature Survey

This section discusses those articles from literature dealing with concurrent program schemata, dependence graphs, sequencing primitives and protocols, state restoration and recovery and related performance issues which contributed directly to the conceptual basis of this research.

Concurrent Program schemata

Of the numerous models of concurrent programs, that of [KAR69] closely matches the one used in the discussion of a general-purpose balanced sequencing protocol with two important differences:

1) The schema of [KAR69] models data as a set M of shared memory locations, each of which contains a value. In its stead, we have data, a set of <name, value, version number> associations. In effect, this approach-

   - abstracts away the issues of data organization such as shared access vs. replication and
   - makes it convenient to discuss state restoration.

   If it were necessary to explicitly deal with issues of data organization, one may always do so by employing appropriate naming conventions. For example, to explicitly represent a technique for updating a replicated datum, the name of the datum may be extended to generate a unique name for each replica, and the extended names used in the program.

2) The [KAR69] schema employs a function G which determines the "outcome" of an event. This "outcome" corresponds to the O_Entry created in SOT at the termination of an event. The difference is that the domain of G is the invocation domain of the step in [KAR69] as opposed to the total domain of the step in the general-purpose sequencer. The two approaches can be proven equivalent. However, the approach employed in the general-purpose sequencer seems to better reflect conventional thinking in programming in that conditional branches are based on the state at the termination of the previous step rather than on the state just prior to it.

Dependence Graphs

The first part of [KUC81] defines five classes of dependence relations (loop, output, anti, flow and input) which have an interesting relationship to the three classes (MM, MI and IM for Modifier-Modifier, Modifier-Invoker and Invoker-Modifier) employed in the general-purpose sequencer. [KUC81] models "Fortran" programs consisting exclusively of assignment statements, For loops and While loops as dependence graphs and discusses compile time optimizations on such graphs. The interesting section in the context of this research is section 2.2 which defines the classes of dependences.

The loop dependence relates a statement to a loop within which it is nested. For this purpose, each loop is identified by a "header", an "increment counter" statement for For loops and a "compute predicate" statement for a While loop. Directed arcs from each header to each statement in the loop (including headers of other loops) represent the loop dependences. Since, in the context of the proposed research, there is no occasion to subject loops to any special treatment, this dependence relation seems unnecessary for the purpose of this research.

The output dependence is a dependence between statements (not necessarily successive) which modify the same datum. Output dependences can be derived from MM, MI and IM dependences as follows:

An output dependence is a path in the dependence graph defined by the regular expression (MM + MI.IM).(MM + MI.IM)* where all arcs represent the same datum.

The antidependence is a dependence from statements which invoke a datum to those which subsequently modify it. Antidependences can be derived from MM, MI and IM dependences as follows:

An antidependence is a path in the dependence graph defined by the regular expression IM.(output dependence)* where all arcs represent the same datum.

The flow dependence is a dependence from a statement which modifies a datum to the next one to invoke it. This is the MI dependence.

The input dependence is a dependence between two statements which invoke the same datum. Since it does not represent a sequencing constraint, it is not of relevance to this research.

Sequencing Primitives and Protocols

Distributed sequencing belongs to the more general class of **distributed decision making** problems. Central to any decision mechanism are:

- A set of rules to map the knowledge of state onto decisions
- In a distributed system, a protocol for the dissemination of such knowledge.

- If the decisions involve any optimism, validation and recovery mechanisms to determine and recover from consequences of incorrect decisions.

"State" in the context of sequencing decisions is commonly termed "control state".

Time and State in Distributed Systems

Several techniques applied to distributed computing employ time-stamps in some capacity. [LAM78] discusses an artificial notion of time in the context of distributed computation which preserves causal relationships. For sequential processes communicating via FIFO channels, events within each process are totally ordered in the sequence in which they occur. The only ordering relation between events of different processes is that the sending of a message occurs before its receipt. To ensure this, a process advances its clock if necessary upon receiving a message. Any other orderings are those derivable from the above. Further, by making time values at each process unique without changing the relative order of events as described above (for example, by appending a unique "process id" to the "time" at each process), a total order consistent with the partial order defined above can be derived. The [LAM78] ordering corresponds to the ordering defined in this proposal as follows:

- The proposed partial order of events of each synchronous component of computation based on the events' use of data is consistent with (and weaker than) Lamport's total ordering of events of the same process.

- If channels are viewed as (the only) data shared by processes with "send" and "receive" as modifying operations on them, the proposed ordering relationships between events of different synchronous components of computation correspond exactly to Lamport's ordering relationships between events of different processes.

[CHA86] presents a notion of global state in the context of distributed computing. The paper presents an algorithm for determining a global state of the computation. By using distributed snapshots which represent such global states as checkpoints, an upper bound of two checkpoints at each process during validation and one during other times can be achieved. A snapshot is initiated by any process which records its state and sends a marker on each of its output channels. The receiver of a marker, if it has not previously participated in the snapshot records its state, records the state of the channel as empty and propagates the marker on its output channels. If the receiver has previously participated in the snapshot, it records the state of the channel as the list of messages received on that channel since the process recorded its state. The set of process and channel states thus recorded constitute a distributed snapshot which can serve as a check-point.

General purpose Distributed Decision Protocols

[SCH82]'s protocol is a general purpose pessimistic solution for any deterministic distributed decision making problem. The protocol works as follows:—

- All relevant information is broadcast on time-stamped messages. All messages broadcast from a site are received at other sites in the order in which they are broadcast. Time-stamps are based on a Lamport-clock [LAM78] mechanism.

- The recipient of a broadcast broadcasts acknowledgement of its receipt.

- The sequence of fully acknowledged messages at a process with no intervening partially acknowledged messages represent the process' knowledge of the state. All the process' decisions are based on that knowledge.

The basic idea is that the fully acknowledged message sequence develops identically at each process. A state transition corresponds to an inclusion of a message in this sequence. Since each process has an identical view of state transition, this approach is in effect equivalent (in behavior, not in performance) to centralization of decision making.

[JEF85]'s protocol is a general purpose optimistic solution for any deterministic distributed decision making problem. Adapting the protocol to [SCH82]'s model of computation, the differences between the two are:—

- No acknowledgement of messages

- When a message is received with a larger time-stamp than that of previous messages, it is optimistically assumed that no messages with intervening time-stamps are assumed. The receipt of each such message represents a state transition.

- When a message with a smaller time-stamp than that of the previous message is received, rollback is initiated to the state prior to the receipt of the earliest message with a time-stamp larger than that of the message just received. Anti messages are sent to negate the messages sent since that state.

- When an anti message is received, if the corresponding message has not yet been received, the two annihilate each other. If the corresponding message has been received, the receiver rolls back to the state prior to the receipt of the negated messages and in turn, sends out anti-messages as described above.

Database and Multi-programming Protocols

[REE79] formulates an abstraction of data which is central to the discussion of rollback in the context of this research. In addition to the common **name** and **value** attributes, a third **create-time** attribute is associated with each datum. Thus, each datum is **named** and has a sequence of **values** ordered by their **create-time**. Other attributes of data employed in [REE79] -viz. read-time and commit-record are not relevant in the context of this research. The attribute

corresponding to **create-time** associated with data in this research is the **version number**. The protocol consists of having each step of the computation choose an appropriate versions of the data needed based on the "pseudotime" time-stamp of the transaction, the create and read times of the sequence of versions and the state of the commit record. The failure to find the appropriate version of any required datum is the heuristic to determine the existence of a sequencing fault and recovery consists of marking the commit records of all versions created by the transaction as "failed" and restarting the transaction.

A lot of research into sequencing concurrent computation has been done in the context of sequencing concurrent transactions on databases. Several strategies were discussed in the context of centralized databases, but they can be adapted to distributed databases.

The earliest proposed primitives applicable to the solution of sequencing problems were proposed as synchronization primitives in the context of multi-programming. Examples of these are the atomic memory fetch and the atomic Test-and-Set .

[ESW76] proposed two phase locking for synchronizing transactions in a database to achieve serializability. The protocol calls for transactions to "lock" data as needed and "unlock" them when a) they are not needed and b) no further data is needed. Data locked by a transaction can not be used by any

other transaction until it is unlocked. This approach is subject to deadlock since it is possible to reach a state in which several transactions wait for each other to unlock data. Two of the several refinements of this protocol are 1) improving data availability via shared locks [ESW76] and 2) avoiding deadlock by pre-ordering entities [SIL80].

Commit based strategies for synchronizing database transactions employ the comparison of time-stamps as the synchronization primitive [KUN81]. The basic theme is to associate a "commit phase" with each transaction. During the commit phase, there is no interaction with other transactions. Prior to the commit phase, the execution of the transaction is unrestricted. The successful end of the commit phase marks the termination of the transaction. If the commit phase is unsuccessful, the transaction has no effect on the database and can be re-started. [KUN81] discusses a commit protocol with the following key features:

- All transactions go through a read phase un-hindered.
- Upon completion of the read phase, each transaction acquires a unique "transaction number".
- A transaction T_j with transaction number t(j) is "validated" if for all T_i with transaction number t(i)<t(j), either of the following conditions are satisfied: (from the paper)

- Ti completes before Tj begins.
- Ti does not write anything read by Tj and completes before Tj begins its write phase.
- Ti does not write anything read or written by Tj and completes its read phase before Tj completes its read phase.

- A validated transaction (by the above rules) executes a write phase which serves as the protocol's commit phase. When condition 3 holds, write phases of those transactions may progress concurrently. A transaction which fails to validate is aborted and re-tried.

This protocol's position on the optimistic-to-pessimistic spectrum is discussed in chapter 6.

Validation

When any optimism is involved in sequencing, some means of validating the execution sequence is required. Most proposed techniques involve comparison of time-stamps associated with events. Representative examples are [JEF85], [KUN81] and [REE79] described above. In all of these cases, all faulty sequences are detected but some correct sequences may be determined to be faulty. This is the price paid for the low computing overhead associated with validation.

A close relative of the validation scheme of the general-purpose sequencer is the approach to run-time validation of programs employed in the Gypsy project [GOO1, 2, 3]. Programs that are difficult to verify at compile-time are executed and logged and the execution validated by matching the log against the specification of the program. In this context:

- the role of the program in the sequencing protocol is that of the specification of a program in Gypsy.
- The role of the triggering mechanism of the sequencing protocol is that of the program in Gypsy.

Rollback and Recovery

This subject has been studied extensively in the context of tolerance to "crashes". Since a crash is conceptually the compromising of a computation's state, and sequencing faults also have such an effect, crash-recovery techniques are applicable to recovery from sequencing faults.

[WOO80] discusses an implementation of a recovery mechanism. Each process establishes checkpoints at arbitrary points of the computation. The issue of checkpointing intervals is not addressed. Associated with each checkpoint, a process maintains a list (the "prop-list") of processes to which rollback will be propagated if that process has to roll back to that checkpoint and a list (the "PRI-list") of processes which could initiate recovery to that checkpoint. Prop-lists are used to propagate a roll-back to other processes to

which messages were sent since establishing the checkpoint. PRI lists are used to determine checkpoints which are candidates for garbage collection.

[RUS80] addresses the problem of determining the checkpointing interval required to ensure that the "domino effect" can not take place. The domino effect is the phenomenon of cascading roll-back feeding back to a process causing it to undo even more of its work. [RUS80]'s model of communication employs message-lists which are a generalization of the conventional channels in that several processes can send to, or receive from any one message-list. For the case where the topology of the communication graph is not restricted but each message-list serves only one sender and one receiver, it is proven that establishing a checkpoint before each receive that was preceded by a send ensures freedom from the domino effect.

The checkpointing and recovery protocol of [KOO87] allows a trade-off between checkpointing intervals and the extent of cascading rollback. It requires processes to keep only the last two checkpoints established and ensures that processes do not have to rollback past the earlier of these checkpoints. The interval between the establishment of checkpoints determines the possible extent of cascading rollback. The establishment of checkpoints is coordinated to ensure that each new set of checkpoints is a distributed snapshot [CHA86] of the system's state.

Performance considerations

[CHA75] derives analytical results for optimal checkpointing intervals under three sets of assumptions. For the most general case analyzed, the assumptions are:

- Poisson distributed fault-detections
- Reprocessing time is proportional to the number of transactions in the recovery region
- Time to process transactions which arrive during establishment of checkpoints or during recovery is negligible in comparison with the Mean Time Between Failures.
- System availability given optimal checkpointing intervals is high.

For the simplest case analyzed, it is also assumed that no errors occur during recovery and that transactions arrive at a constant rate. The intermediate case does not assume the absence of errors during recovery. [CHA72] contains a survey of analytical models of rollback and recovery.

[GEL76] determines the maximum transaction load, response time for a given transaction load and time-overhead of recovery as a function of failure rate. The main difference from [CHA75] is that transaction arrivals during establishment of checkpoints or during recovery are not ignored.

Chapter 7
Summary of Results and Conclusions

This dissertation describes, formalizes and analytically evaluates balanced sequencing protocols, a new class of protocols for sequencing distributed computations.

Results

Two approaches to the problem are considered. The first of these is a general-purpose transformation of any given protocol into a spectrum of balanced protocols. While the existence of such a transformation proves that theoretically, any protocol can be generalized to a spectrum of balanced protocols, the complexity of sequencing decisions is rather high. This suggests the need for the second approach— efficient, problem-specific spectra of balanced protocols.

Problem-specific protocols are proposed and analytically evaluated for the producer-consumer problem, distributed semaphores, distributed locking and echo-damping.

For the producer-consumer problem, it was assumed that the times to perform the operations **Produce**, **Consume**, **Read_Shared_Memory** and **Write_Shared_Memory** are time-invariant. The chosen model also does

not penalize the additional memory required when optimism is introduced. Under these simplifying assumptions, the optimal balance is derived as a function of the speed of the above operations.

For a distributed implementation of semaphores and multi-modal locks, the optimal balance is derived as a function of the availability of the semaphore/lock and the expected time-cost of recovery. For the examples considered, **P** (or **lock**) operations are typically speeded up by a few message-delays. The distributed database example illustrates that the benefits of balanced sequencing are sensitive to recovery costs.

The **echo phenomenon**, described in chapter 5, is a performance bottle-neck for optimistic protocols. The performance deterioration it causes is analytically determined for some simple networks of processes. A variant of balanced sequencing is proposed to counter-act this phenomenon.

Practical considerations

Several practical factors must be considered when implementing balanced sequencing protocols. The following is a discussion of some of these problems.

For the examples considered, the optimal balance between optimism and pessimism was analytically determined from simple models of system behaviour. In the interest of simplicity and analytical tractibility, some factors of practical concern were not incorporated into the model. For example,

balanced sequencing protocols are usually more complex than either optimistic or pessimistic protocols. This increase in complexity impacts run-time overhead because of factors such as increased demand for resources and increased paging activity.

Where the model of system behaviour is not analytically tractible, simulation studies may be necessary. An alternative to simulation is to tune the balance between optimism and pessimism adaptively; i.e., by varying the balance based on observation of past behaviour. Adaptive tuning may be considered for systems whose behaviour can be economically monitored and tends to be repetitious.

In studying the echo phenomenon, a simple model which assumed constant progress and recovery rates was employed. In practice, where these rates will be more random and time-variant, the echo phenomenon can be expected to exhibit a much more complex behaviour. It is hoped that a simulation study currently being planned will shed more light on this problem.

Future work

It is hoped that this dissertation will encourage further research in the field of balanced sequencing. The analytical methods employed in the examples do not easily generalize to more complex examples. There is a need for better analytical techniques and detailed simulation models. An actual implementation of a balanced sequencing protocol and a study of its benefits and weaknesses when applied to the execution of "real-world" distributed programs is required

before balanced sequencing is seriously considered as a viable alternative to conventional approaches.

Conclusions

In conclusion, the performance of sequencing protocols can be significantly improved by implementing them as a spectrum of Balanced Sequencing Protocols and customizing them by choosing the appropriate level of balance based on the characteristics of the system. The research reported in this dissertation has established that balanced sequencing protocols have a substantial potential for enhancing the performance of distributed systems. Whether or not this potential can be realized remains to be determined by more in-depth simulation modeling and implementation studies. Extreme optimism as proposed by Jefferson [JEF85] should be used with caution as it is subject to potentially severe performance degradation caused by the Echo Phenomenon.

Bibliography

[AAH87] Y. Aahlad, J.C. Browne, "Balanced Protocols for Sequencing Distributed Computations" Technical Report TR-87-39, University of Texas at Austin, Department of Computer Sciences, October 1987.

[AAH89] Y. Aahlad, J.C. Browne, "Balanced Sequencing Protocols" Proceedings of the SCS Multiconference on Distributed Simulation, Tampa, FL, March 1989, 58-63.

[AGE77] T. Agerwala, "Some Extended Semaphore Primitives" Acta Informatica 8, 201-220, 1977

[AGR85] R.Agrawal, D.J. Dewitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation" ACM Trans. DataBase Systems, December '85

[BER87] P.A. Bernstein, V. Hadzilacos, N. Goodman, "Concurrency Control and Recovery in Database Systems" Addison-Wesley, Reading, Mass., 1987.

[BRI73] P. Brinch-Hansen, "Operating System Principles" Prentice-Hall, Englewood Cliffs, New Jersey, 1973

[CAR88] M.J. Carey, M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication" Proc. 14th VLDB Conf., Los Angeles, CA, 1988, 13-25

[CHA72] K.M. Chandy,C.V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs" IEEE TOCS C21(6), June 1972

[CHA75] K.M Chandy, J.C. Browne, C.W. Dissly,W.R. Uhrig, "Analytical Models for Rollback and Recovery Strategies in Data Base Systems" IEEE Trans. S. E, SE-1(1), March 1975

[CHA81] K.M Chandy, J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations" CACM 24 4):198-206, April 1981

[CHA86] K.M. Chandy, L. Lamport, "Distributed Snapshots : Determining Global States of Distributed Systems" ACM TOCS 3(1):63-75, February 1985

[DAS90] P. Dasgupta, Z.M. Kedem, "The Five Color Concurrency Control Protocol: Non-Two-Phase Locking in General Databases" ACM TODS 15 (2):281-307, June 1990

[DIJ68] E.W. Dijkstra, "Co-operating Sequential Processes" Programming Languages: NATO Advanced Study Institute:43-112, Academic Press, London, F. Genuys (ed.) 1968

[DIJ76] E.W.Dijkstra, "A Discipline of Programming" Prentice Hall, Englewood Cliffs, New Jersey, 1976

[DIJ80] E.W. Dijkstra, C.S. Scholten, "Termination Detection for Diffusing Computations", Information Processing Letters 11(1):1-4, August 1980

[ESW76] K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System" CACM 19(11):624-633, November 1976

[FUJ88] R.M. Fujimoto, "Performance Measurement of Distributed Simulation Strategies" Proceedings of the SCS Multiconference on Distributed Simulation, San Diego, CA, February 1988, 14-20

[GAR83] H. Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database" ACM TODS 8 (2):186-213, June 1983

[GEL76] E. Gelenbe, D. Derochette, "Maximum Load and Service Delays in a Data-Base System with Recovery from Failures" Modelling and Performance Evaluation of Computer Systems, N. Holland Pub. Co., New York. H. Beilner & E. Gelenbe, ed., 1976

[GOO1] D.I. Good, R.M. Cohen, J. Keeton-Williams, "Principles of Proving Concurrent Programs" Instt. for CS, UT Austin ICSCA-CMP-15, January 1979

[GOO2] D.I. Good, "The Proof of a Distributed System in Gypsy" Instt. for CS, UT Austin Tech. Rprt 30, September 1982

[GOO3] D.I. Good, R.M. Cohen, "Verifiable Communications Processing in Gypsy" Instt. for CS, UT Austin, ICSCA-CMP-11June 1978

[GRA75] J.N. Gray, R.A. Lorie, G.R. Putzolu, I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base" Research Report RJ1654, IBM, September 1975

[HOA85] C.A.R Hoare, "Communicating Sequential Processes" Prentice/Hall International, UK, 1985

[JEF85] D. Jefferson, "Virtual Time" ACM TOPLAS 7(3):404-425 July 1985

[JOH88] D. Johnson, W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing" Proceedings of the 7th ACM Symposium on Principles of Distributed Computing, pages 171-181, 1988

[KAR69] R.M. Karp, R.E. Miller, "Parallel Program Schemata" Journal of Computer and Sys. Sciences, 1969

[KOO87] R. Koo, S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems" IEEE Transactions on Software Engineering, 13(1):23-31, January 1987

[KOR83] H.F. Korth, "Locking Primitives in a Database System" JACM 30 (1):55-79, January 1983

[KOR90] H.F. Korth, E. Levy, A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions" Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane, pages 95-106, August 1990

[KUC81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, M. Wolfe, "Dependence Graphs and Compiler Optimizations" POPL, January 1981

[KUN81] H.T. Kung, J.T. Robinson, "On Optimistic Methods for Concurrency Control" ACM TODS 6(2):213-226, June 1981

[LAM78] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System" CACM 21(7):558-565, July 1978

[LAM79] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs" IEEE TOC 28(9):690-691, September 1979

[LAM83] L. Lamport, "Solved Problems, Unsolved Problems and Non Problems in Concurrency" Invited Address, PODC 1983, appeared in print in conf. proc. of 1984

[MAE85] M. Maekawa, "A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems" ACM TOCS 3(2):145-159, May 1985

[PAP81] C.H. Papadimitriou, "On the Power of Locking", Proc. ACM SIGMOD Int'l Conf. on Management of Data, pages 148-154, Ann Arbor, MI, April 1981

[PAP86] C.H. Papadimitriou, "The Theory of Database Concurrency Control", Computer Science Press, Rockville Md, 1986

[REE79] D.P. Reed, "Implementing Atomic Actions on Decentralized Data" Sigops, 1979

[REE79] D.P. Reed, R.K. Kanodia, "Synchronization With Eventcounts and Sequencers" CACM 22(2):115-123, February 1979

[RIC81] G. Ricart, A. Agarwala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks" CACM 24: 9-17, 1981

[RUS80] D.L. Russell, "State Restoration in Systems of Communicating Processes" IEEE Trans. SE, SE-6(2), March 1980

[SCH82] F.B. Schneider, "Synchronization in Distributed Programs" ACM TOPLAS 4(2):125-148, April 1982

[SIL80] A. Silberschatz, Z. Kedem, "Consistency in Hierarchical Database Systems" JACM 27(1):72-80, January 1980

[STR85] R.E. Strom, S. Yemini, "Optimistic Recovery in Distributed Systems" ACM TOCS, 4(3):204-226, August 1985

[THO79] R.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases" ACM TODS, 4(2):180-209, June 1979

[WOL88] S. Wolfram, "Mathematica™, A System for Doing Mathematics by Computer" Addison Wesley, Redwood City, California, 1988

[WOO80] W.G. Wood, "Recovery Control of Communicating Processes in a Distributed System" Technical Report 158, University of Newcastle Upon Tyne, Computing Laboratory, November 1980

VITA

Yeturu Aahlad was born in Madras, India on March 11, 1958, the son of Yeturu Venkata Satyasena Reddi and Yeturu Saroja Reddi. In July 1975, he entered the Indian Institute of Technology in Madras, India. In August 1980, he received the degree of Bachelor of Technology in Electronics and Communication from the Indian Institute of Technology. Starting July 1980, he worked for one year as a Development Engineer at Yamuna Digital Electronics, Inc. in Hyderabad, India. In September 1981, he entered the Graduate School of the University of Texas at Austin. In December 1983, he received the degree of Master of Science in Engineering from the University of Texas. From January 1984 to August 1990, he worked as a Teaching and/or Research Assistant for the Department of Computer Science at the University of Texas. From September 1990, he worked as a Staff Member at IBM's Palo Alto Scientific Center.

Permanent address: 801 Foster City Boulevard, #105
Foster City, California 94404

This dissertation was typed by Yeturu Aahlad using Microsoft Word on a Macintosh computer. MacDraw, MacEqn and Microsoft Excel were also used in the preparation of this dissertation.