Commit Processing in Distributed Secure and Real-Time Transaction Processing Systems

A Dissertation
Submitted in Partial Fulfilment of the
Requirements for the Degree of
Master of Engineering
in Computer Science and Technology
(Bangalore University)

By
P. Kavitha Nagaraj

Guided By

Mr. Jayasurya Venugopalan	Dr. Jayant Haritsa
Asst. Professor	Asst. Professor
BMSCE, Bangalore-19	IISc, Bangalore-12

DEPARTMENT OF COMPUTER SCIENCE
BMS COLLEGE OF ENGINEERING
BANGALORE – 560 019, INDIA

October 1998

Commit Processing in Distributed Secure and Real-Time Transaction Processing Systems

A Dissertation
Submitted in Partial Fulfilment of the
Requirements for the Degree of
Master of Engineering
in Computer Science and Technology
(Bangalore University)

By
P. Kavitha Nagaraj

Guided By

Mr. Jayasurya Venugopalan
Asst. Professor
BMSCE, Bangalore-19

Dr. Jayant Haritsa
Asst. Professor
IISc, Bangalore-12

DEPARTMENT OF COMPUTER SCIENCE
BMS COLLEGE OF ENGINEERING
BANGALORE – 560 019, INDIA

October 1998

Bangalore University

BMS College of Engineering

BANGALORE – 560 019.

CERTIFICATE

This is to certify that the project work entitled

DISTRIBUTED COMMIT PROCESSING IN SECURE REAL TIME DATABASES

has been successfully carried out by

P. KAVITHA NAGARAJ

in partial fulfilment for the award of the degree of

Master of Engineering

in

Computer Science and Technology

of Bangalore University in the academic year 1998

Guide and H.O.D.
Mr. Jayasurya Venugopalan
Asst. Professor
BMSCE, Bangalore-19

P.G. Co-ordinator
Dr. A. Sreenivasan
Professor
BMSCE, Bangalore-19

To

My Beloved Ones

Abstract

Database systems are among the most important applications of computer technology. They provide versatile methods for storing and dealing with large volumes of data. A *Distributed database* is a database where different objects may be stored at different sites and where the users may issue transactions at any site in the system. Distributed database systems implement a transaction *commit protocol* to ensure transaction atomicity. A commit protocol guarantees the uniform commitment of distributed transaction execution, that is, it ensures that all the participating sites agree on the final transaction outcome (commit or abort). Most importantly, this guarantee is valid even in the presence of site or network failures. Over the last two decades, a variety of commit protocols have been proposed by database researchers. These include the classical *two phase commit (2PC)* protocol, its variations such as *Presumed Commit* and *Presumed Abort* which have the 2PC protocol streamlined so as to reduce the number of messages exchanged, and then there is *three phase commit* protocol which is an improved version of 2PC which helps prevent blocking at the expense of extra messages and logging of records. These commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites where the distributed transaction executed. In addition, several log records are generated, some of which have to be "forced", that is, flushed to disk immediately. Due to these costs, commit processing can result in a significant increase in transaction execution times, and therefore the choice of commit protocol becomes an important decision in the design of a distributed database system.

Real-Time Database (RTDB) systems cater to applications wherein transactions have specific *timing constraints*, usually in the form of completion *deadlines*, and the goal of the system is to meet these deadlines, that is, to process transactions before their deadlines expire. Such real-time applications include aerospace and military systems, computer integrated manufacturing, stock markets, etc. Many real-time database applications arise in safety-critical installations and military systems where enforcing **security** is crucial to the success of the enterprise. Unfortunately, the mechanisms for achieving security and for meeting transaction deadlines often work at cross-purposes, making it difficult to design efficient secure RTDBS. Many of these applications are inherently distributed in nature. We investigate here the performance implications of supporting transaction atomicity in secure distributed real-time database systems where a set of transactions are assigned classification levels and are allowed to access data of a specific clearance level according to the Bell-LaPadula model. And in particular, we consider applications with "firm-deadlines"—for such applications, completing a task after its deadline has expired is of no utility and may even be harmful. Therefore, transactions missing their deadlines are immediately discarded from the system without being executed to completion. The performance goal in a firm-deadline RTDB system is to minimize "deadline miss percent", that is, the percentage of transactions missing their deadlines.

Using a detailed simulation model of a distributed secure firm-deadline real-time database system, we profile the real-time performance of a representative suite of previously proposed commit protocols that are customized for the real-time domain. We also evaluate the commit protocol, called **RT-OPT**,that allows transactions to "optimistically" borrow uncommitted data in a controlled manner. The RT-OPT protocol is easy to implement and incorporate in current systems, and can coexist with most of the other optimizations proposed earlier. For example, OPT can be combined with current industry standard commit protocols such as Presumed Commit or Presumed Abort. It incorporates optimizations that cater to the special features of the real-time domain.

The results obtained from these simulation experiments are : First, the choice of commit protocol clearly affects the magnitude of the influence on the deadline miss percentage. Second, the new real-time optimistic commit protocol, RT-OPT, provides the lowest deadline miss percentage among the real-time commit protocols evaluated in our study, for a variety of workloads and system configurations.

In summary, having integrated security with real-time database system, we profile the performance of a representative suite of previously proposed commit protocols that are customized for the real-time domain and from the results it is observed that RT-OPT exhibits better real-time performance than all of the standard real-time commit protocols evaluated in our study.

Acknowledgements

It is indeed a pleasure to express my sincere thanks to Professor S. R. Krishnamurthy, P.G. Co-ordinator Dr. A Sreenivasan, and principal Dr. Haranath for the encouragement and support they have given me.

There are not enough words to express my gratitude to Mr. Jayasurya Venugopalan our Head of the Department at B.M.S. College of Engineering who has always motivated me to do better. He has always been kind enough to listen and advice whenever I had faced difficulties during my project work.

Prof. Balakrishnan, Chairman of SERC has been very kind to have taken the time to consider my request and for having suggested Dr. Jayant Haritsa as the right person to be approached. Thanks to him for having allowed me to do my project at SERC.

I would like to express my gratitude to Dr. Jayant Haritsa who gave me the wonderful oppurtunity to work on this project under his guidance. He has constantly encouraged independence of thought, and to explore and learn new things. And whenever I found something difficult to understand in the subject he would always make time and explain the topic in a very easy to understand manner. It was a pleasure working in his fine lab.

Many thanks to Manjusha, Maya, Vikram and Binto for bearing with my constant stream of doubts in UNIX, and in the process of which I have gained their friendship and understood UNIX better.

Due to the support and co-operation of my colleagues and friends D.N. Sujatha, Lakshmi Narayan, Poornima, K.S. Sujatha, Ramesh, and Umesh I was able to concentrate much better on my project work. Sincere thanks to all of them .

Honestly the fact that I have been able to complete my project and present this thesis is only due to the constant encouragement, love and patience of my husband, parents, sister and my brother. And last but not the least is my adorable son, whose liveliness would make me relax during the difficult stages of my work here.

Contents

List of Figures

List of Tables

CHAPTER

1

Introduction

1.1 Background

A **distributed database system** consists of a collection of **sites**, connected together via some kind of communications network, in which each site is a database system site in its own right, but the sites have agreed to work together so that a user at any site can access data anywhere in the network exactly as if the data were all stored at the user's own site. The distributed database system can thus be regarded as a kind of **partnership** among the individual local DBMSs at the individual local sites; a new software component at each site - logically an extension of the local DBMS-provides the necessary partnership functions, and it is the combination of this new component together with the existing DBMS that constitutes what is usually called the **distributed database management system**

A database is a collection of objects which satisfy a set of integrity constraints. For example, in a bank, an integrity constraint may be that an account cannot have a negative balance. Users interact with the database by executing programs that perform the functions desired by the users. The execution of such programs results in a partially ordered set of read and write operations. This set of operations is called a **transaction**. A transaction terminates by either executing a *commit* or an *abort* operation. The primary feature of a transaction is that it is an *atomic* unit of work that is either completed in its entirety or not done at all. A commit opertion implies that the transaction was successful, and hence all its updates should be incorporated into the database. An abort operation indicates that the transaction has failed, and hence requires the database management system to cancel or abolish all its effects on the database system. In short, a transaction is an "all or nothing" unit of execution. A transaction that updates the objects of the database must preserve integriry constraints of the database. Transfer of money from one account to another, reservation of train tickets, filing of tax returns, entering marks on a student's grade sheet, are all examples of transactions.

For a variety of reasons, many database applications store their data *distributed* across multiple sites that are connected by a communication network. In this environment, a single transaction may have to execute at many sites, based on the locations of the data that it needs to process. A potential problem associated with distributed transaction execution is that some sites could decide to commit the transaction while the others could decide to abort it, resulting in a violation of transaction atomicity. To address this problem, distributed database systems use a transaction **commit protocol**. A commit protocol ensures the uniform commitment of the distributed transaction, that is, it ensures that all the participating sites agree on the final outcome (commit or abort) of the transaction. Most importantly, this guarantee is valid even in the presence of site or network failures.

Over the last two decades, a variety of distributed transaction commit protocols have been proposed by database researchers. To achieve their functionality, these commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites (where the distributed transaction executes). In addition, several log records are generated, some of which have to be "forced", that is, flushed to disk immediately in a synchronous manner. Due to these costs, commit processing can result in a significant increase in transaction execution times, making the choice of commit protocol an important design decision for distributed database systems.

From a performance perspective, commit protocols can be compared on the following three issues:

Effect on Normal Processing: This refers to the extent to which the protocol affects the normal (no-failure) distributed transaction processing performance. That is, how expensive is it to provide atomicity using this protocol?

Non-blocking: A commit protocol is said to be *non-blocking* if, in the event of a site failure, it permits transactions that were being processed at the failed site to terminate at the operational sites without waiting for the failed site to recover. With blocking protocols, there is a possibility of transaction processing grinding to a halt in the presence of failures (as explained in Section 3.5). Non-blocking protocols, on the other hand, are designed to ensure that such major disruptions do not occur. To achieve their functionality, however, they usually incur additional messages and force-writes than their "blocking" counterparts.

Recovery: The issue of speed of recovery appears less critical for two reasons: First, failure durations are usually orders of magnitude larger than recovery times. Second, failures are usually rare enough that we do not expect to see a difference in *average* performance among the protocols because of one commit protocol having a faster recovery time than the other. With this viewpoint, we focus here on the *mechanisms* required during normal operation to provide the functionality of recovery, rather than on the recovery *process* itself.

1.2 Motivation

Imagine the situation where an alien bomber could strike a critical target within ten minutes of entering your country's airspace, unless a pre-emptive mission is successfully launched and completed. The pre-emptive mission will involve the following three phases: The first phase consists of analyzing the data from the air force database and the numerous surveillance sensors mounted all around. The second phase involves preparation of a suitable defense plan based on the data analysis. In the third phase, the prepared plan is executed before the expiry of the time limit, that is, within ten minutes. You, as the commander of defense operations, are facing a problem of *three dimensional* complexity: (1) There is an enormous amount of data to be consulted, processed and updated, and while doing so the *database correctness* should be maintained; (2) There is a *time con-*

straint for the task — the mission must be completed within ten minutes; (3) During every stage of the mission, *data security* must be ensured — giving unauthorized access to the secret military information may be catastrophical. In this thesis we focus on the design of information systems that can simultaneously and effectively meet these three challenges, that is, on the design of **secure realtime database systems (SRTDBS)**.

1.3 Real Time Databases

Applications that deal with large quantities of information typically use database systems for efficient data storage, processing and retrieval. For data-intensive applications that are faced with timing requirements, the ability of the underlying database system to satisfy transaction timing constraints becomes a key factor in determining both the feasibility of the application and the extent of its use [7]. A system that is specifically designed for efficiently processing such transactions is referred to as a **Real-Time Database System (RTDBS)**. The objective of the system is to meet the transaction deadlines, that is, to process transactions before their deadlines expire.

Many RTDBS applications arise in safety-critical installations and military systems where enforcing **security** is crucial to the success of the enterprise. Given that the US government itself is classifying about seven million new pieces of information every year [5], no further evidence is required for emphasizing the importance of information security. Surprisingly, however, the issue of providing security in RTDBS has received comparatively little attention although real-time database research has been underway for close to a decade now (see [25] for a survey). In our study, we partially address this lacuna by making a detailed investigation of the mechanisms for providing security in the context of real-time applications with "firm-deadlines" [8].

For designing commit protocols for a distributed RTDB system, we need to address three major questions: First, how do we adapt the standard commit protocols into the real-time domain? Second, how to integrate security into the real time domain? and Third, how do the real-time and secure variants of the commit protocols compare in their performance? We address the last two questions questions in this thesis based on the work done to address the first question [28]. In particular, we consider applications that have *firm-deadlines*. For such applications, completing a transaction after its deadline has expired is of no utility and may even be harmful. Therefore, transactions that miss their deadlines are "killed", that is, immediately aborted and discarded from the system without being executed to completion. Accordingly, the performance metric is *the percentage of missed transaction deadlines*.

1.4 Security Violations

Imagine the situation wherein an RTDBS contains both "secret" and "public" data. Security violations in this RTDBS can occur if, for example, information from the "secret" database is transferred by corrupt high security transactions to the "public" database where they are read by conspiring low security transactions. Such *direct* violations can be eliminated by implementing the classical Bell–LaPadula security model [32] which imposes restrictions on the data accesses permitted to transactions, based on their security levels. The Bell–LaPadula model is not sufficient, however, to protect from "covert channels". A covert channel is an *indirect* means by which a high security transaction can transfer information to a low security transaction [33]. For example, if a low security transaction requests access to an exclusive resource, it will be delayed if the resource is already held by a high security transaction, otherwise it will be granted the resource immediately. The presence or absence of the delay can be used as a "signaling" or encoding mechanism by a high security transaction passing secret information to the low security transaction. Note that nothing "obviously illegal" has been done in this process by the conspiring transactions, from the system perspective.

Covert channels can be prevented by ensuring that low security transactions do not "see" high security transactions – this notion is formalized in [6] as *non-interference*, that is, low security transactions should not be able to distinguish between the presence or absence of high security transactions. This can be implemented, for example, by providing higher priority to low security transactions whenever a conflict occurs between a low security transaction and a high security transaction. From a system perspective, it translates to implementing transaction processing components that support the non-interference feature. In this thesis, we address the issue of designing such covert-channel-free components in a manner that minimizes the impact on the real-time performance objectives.

1.5 Design Challenges

An SRTDBS has to *simultaneously* satisfy two requirements, namely, provide security and minimize the number of missed deadlines. Unfortunately, the mechanisms for achieving the individual goals often work at cross-purposes. In an RTDBS, high priority is usually given to transactions with earlier deadlines in order to help their timely completion. On the other hand, in secure DBMS, low security transactions are given high priority in order to avoid covert channels (as described in Section 1.4). Now consider the situation wherein a high security process submits a transaction with a tight deadline in a secure real-time database system. In this case, it becomes difficult to assign a priority since assigning a high priority may cause a security violation whereas assigning a low priority may result in a missed deadline.

In addition to the above issue, there are several related problems that arise while integrating security into the RTDBS framework:

1. Given the paramount importance of security, the database system is forced to assign transaction priorities based primarily on security clearance levels and only secondarily on deadlines. A major problem arising out of this preferential treatment of low security transactions is that of *fairness* – high security transactions usually form a disproportionately large fraction of the transactions with missed deadlines. This is an especially problematic issue because it is the high security transactions, which are supposed to be having higher privileges, that are being discriminated against in favor of the low security transactions.

2. There may be a multitude of covert channel mediums in the transaction processing component, all of which have to be eliminated to make it truly secure. For example,

in the Buffer Manager, the presence or absence of a delay in acquiring a free buffer slot, or the presence or absence of a specific page in the buffer set, or the allocation of a particular (physical) buffer slot, could all be used as channel mediums.

3. Straightforward solutions to eliminate covert channels such as using "static" resource allocation policies where each transaction security class has a pre-assigned set of resources, or using "replication"-based policies wherein multiple copies of data pages are maintained, may result in very poor resource utilization. The goal therefore is to design "dynamic" and "one-copy" policies that are demonstrably secure.

4. Finally, "noise"-based approaches to addressing covert channels, wherein noise is introduced into the signaling channel – by executing *dummy* transactions, for example – are impractical for covert channels that use *data* as the medium (for example, presence or absence of a lock on a pre-determined data item). This is because, unlike physical resources which are typically few in number, the number of data items is usually enormous, especially in a database system. In fact, in heavily loaded systems, noise at the physical resources may be generated "for free", but this will probably never be the case for data since it is trivial to add an additional data item that is of no relevance to other transactions. Therefore, explicitly making data access covert-channel-free is more vital than doing the same for resource access.

In summary, the design of an SRTDBS involves three dimensional complexity as featured in Figure 1.1 since they have to ensure data consistency, enforce security, and provide good realtime performance. Our contributions towards the design of such systems are described in the next section.

Contributions

In our study, we make a detailed investigation of the mechanisms for providing security in the context of real-time applications with "firm-deadlines". And , we quantitatively investigate the performance implications of supporting transaction atomicity in distributed

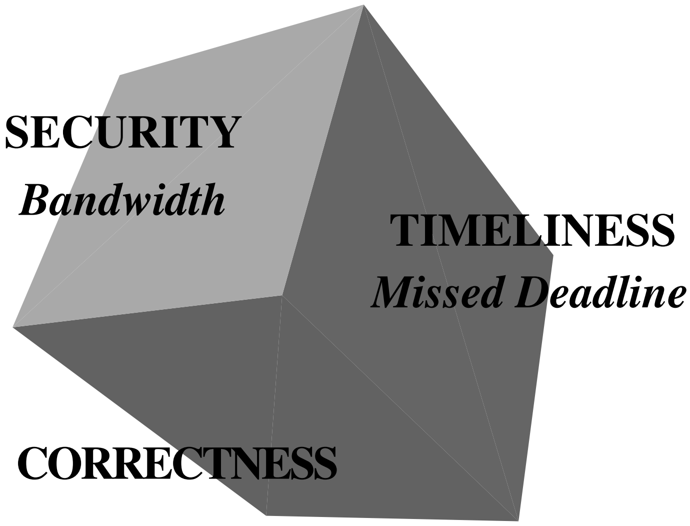

Figure 1.1: Complexity of Transaction Processing

secure RTDB systems by performing an extensive set of simulation experiments. Our contributions are three-fold:

1. We first precisely define the process of transaction commitment and the conditions under which a transaction is said to miss its deadline in a firm-deadline distributed real-time database system. Subsequently, we customize the conventional commit protocols for the firm-deadline secure RTDB environment.

2. Using a detailed open model of a distributed secure firm-deadline real-time database system, we profile the real-time performance of a representative suite of secure commit protocols that are customized for the real-time domain.

The results obtained from these experiments are

- The new real-time optimistic commit protocol, RT-OPT, provides the lowest deadline miss percentage among the real-time commit protocols evaluated in our study, for a variety of workloads and system configurations.

- A non-blocking version of RT-OPT exhibits better real-time performance than all of the standard blocking real-time commit protocols evaluated in our study.

1.6 Organization

An overview of distributed databases, security, RTDB systems is given in this Chapter 1. Details regarding the architecture of secure real time distributed data bases is given in the Chapter 2. A detailed description of a variety of commit protocols is provided in Chapter 3. And we also describe in detail OPT, the newly proposed protocol in [28]. In Chapter 4 we discuss the issues involved in adapting the conventional commit protocols into the real-time secure framework. The real-time commit protocol RT-OPT is also described in this chapter. The simulation model for the distributed SRTDB system is described in Chapter 5, and the results of the experiments are presented in Chapter 6. Finally, in Chapter 7, we provide an overall summary of the work presented in this thesis and identify future research avenues.

Major parts of the material for this thesis has been taken from [30] and [31]

CHAPTER

2

Secure Real Time Distributed Databases

2.1 Introduction

The secure real-time transaction processing domain is a complex union of database technology, real-time technology and information security. The purpose of this chapter is to review the basics of these three areas and then to describe the problems that arise in their combination. We begin this chapter with a discussion of traditional transaction processing and then move on to real-time database systems. Subsequently, we present the issues related to secure transaction processing. Finally, we describe the design, architectural and implementation issues for SRTDBS. **Material for this chapter is taken from [31]**

2.2 Transaction Processing

A database system consists of a collection of interrelated data and a set of programs that access the data [19]. The goal of the database system is to support efficient storage, retrieval and processing of large amount of data. Applications interact with the database system through *transactions*, which constitute the basic unit of work. Transactions provide the so-called **ACID** semantics, described below:

Atomicity: A transaction is an atomic unit of work comprising of many operations. If all the operations have been performed successfully, the transaction commits. If some operation of the transaction fails and the transaction is aborted, the partial results of the transaction are undone.

Consistency: An individual execution of a transaction transforms the database from one consistent state to another consistent state.

Isolation: A transaction does not make its database modifications visible to other transactions before its commitment. This guarantees the standard notion of database consistency called *serializability*, which states that the concurrent execution of a set of transactions should be equivalent to some serial execution of the same set of transactions. That is, the outcome of concurrent execution of transactions, as reflected in the stored data, should be the same as that produced by executing the transactions in some serial order.

Durability: Once a transaction has committed, the effects of the transaction are never lost despite any subsequent system failures.

The database system provides the above semantics through three components working cooperatively, namely, the *Concurrency Control Manager*, the *Buffer Manager*, and the *Recovery Manager*.

2.2.1 Concurrency Control Manager

The CC manager implements a protocol that ensures serializability by resolving concurrent data accesses in a manner that induces a serialization order among the conflicting transactions. Several serialization mechanisms have been proposed, including *locking*, *validation* and *timestamping*, and a rich variety of protocols have been developed based on these schemes.

Two methods, *blocking* and *restarts*, are available for resolving data conflicts, and virtually all CC protocols designed for "single copy" databases (where only a single copy of each data item is maintained) use some combination of these methods. A blocking decision causes one of the conflicting transactions to wait for the other to complete, while a restart decision causes one of the conflicting transactions to be rolled-back.

There are also "multiversion" systems where multiple copies of data elements are maintained in an attempt to reduce the data conflicts. Multiversion CC protocols such as *multiversion timestamp ordering*, *multiversion two phase locking*, etc. are available for ensuring serializability in such systems.

2.2.2 Buffer Manager

In most computer setups, the operating system is responsible for the management of physical resources. However, the policies used by the OS are often unsuitable to meet database objectives. Therefore, some OS functionalities are re-implemented by database system builders to suit their requirements. In particular, database systems reserve a region of main memory to serve as a "database cache" of the persistent data on disk. This memory space is usually referred to as the *buffer pool* and is organized as a set of *slots*, each of which can hold a single data page.

A buffer page that matches the corresponding page on the disk is said to be "clean" whereas a modified page is said to be "dirty". Dirty pages have to be flushed to disk before they can be replaced by other pages. Sometimes, even without the need for replacement, dirty pages have to be "force-written" to disk – for example, at transaction commit time, all buffer pages containing log records related to the committing transaction have to be forced to the disk.

Transactions that intend to access the data in a buffer slot are required to first *pin* the slot in the appropriate mode (read or write). The pin ensures the physical consistency of the page during the access period. Multiple read pins may simultaneously exist on a page but only a single write-pin is permitted at a time. When the transaction completes its current usage of the pinned page, it issues the corresponding *unpin* command. Of course, it may access this same page again later following the pin–unpin procedure as before.

If a requested page is found to be already in the buffer pool, it is termed a "buffer hit", whereas if it has to be brought in from the disk, it is termed a "buffer miss". Buffer managers attempt to utilize the locality typically exhibited in database reference patterns to maximize the number of buffer hits. Three kinds of reference locality are usually observed: *inter-transaction locality* (i.e."hot spots"), *intra-transaction locality* (transaction's internal reference locality), and *restart locality* (restarted transactions usually make the same set of accesses as their original incarnation).

There are three components commonly associated with buffer management: *allocation*, *replacement* and *synchronization*. The allocation algorithm determines the assignment of buffers to transactions. The replacement algorithm determines which page is to be replaced in order to accomodate a new page if no empty buffers are currently available. Lastly, the synchronization algorithm ensures that conflicting pins are not held on a page at the same time.

2.2.3 Recovery Manager

The recovery manager is responsible for ensuring that the database contains all the effects of the committed transactions and none of the effects of aborted ones. Most importantly, this guarantee is valid even in the presence of transaction or system failures.

For *distributed* databases, a *commit protocol* has to be implemented to ensure that *all* sites participating in the distributed execution of the transaction agree on the final decision (commit or abort). A large number of such commit protocols have been designed including the classical *Two Phase Commit (2PC)* protocol, its variations such as *Presumed commit (PC)* and *Presumed Abort (PA)*, and *Three Phase Commit (3PC)*.

2.2.4 Admission Control

A system that is overloaded with transactions may often deliver less than its best possible performance. Therefore, for performance improvement, some systems employ an *admission control* policy that controls the entry of transactions into the system. The decision to admit or deny a transaction into the system is taken based on the current transaction population and the "desired" population in the system. The desired population is determined based on some heuristics and the computation of the desired population is often based on the past history of the system performance. A transaction that is denied admission at the entry point maybe discarded from the system.

2.3 Real-Time Database Systems

Having discussed the basics of database transaction processing, we now move on to **Real-Time Database Systems (RTDBS)**. A Real-Time Database System (RTDBS) is a transaction processing system that is designed to handle workloads where transactions have timing constraints. A time constraint is, typically, expressed in the form of a **deadline**, that is, the application submitting the transaction would like it to be completed before the deadline expires. Accordingly, the objective of the RTDBS is to complete transactions before their deadlines expire. How soon a transaction completes with respect to its deadline is of no relevance at all – what is important is *whether or not* the transaction completes before its deadline. That is, having a transaction complete just before its deadline is no different to having it finish earlier. Therefore, in contrast to a conventional DBMS where the goal usually is to minimize transaction response times, the emphasis in an RTDBS is on meeting transaction deadlines.

Note that an RTDBS has to *simultaneously* satisfy two different kinds of constraints: data integrity constraints and real-time constraints (see Figure 2.1). This *dual* requirement makes real-time transaction scheduling more complex than task scheduling in conventional real-time systems or transaction scheduling in conventional database systems.

Based on how the application is impacted by the violation of time constraints, real-time applications can be grouped into three categories: 1. Hard Deadline; 2. Firm Deadline; and 3. Soft Deadline. These three categories are shown in Figure 2.2; in this figure, transaction T arrives into the system at time A_T with an associated deadline D_T. A transaction that completes successfully before its deadline is *InTime* and fails to complete before the deadline is *Late*. For a hard deadline application, missing a deadline is equivalent to a catastrophe symbolized by the exploding bomb shown in Figure 2.2a. Life-critical applications, such as flight control systems or missile guidance systems, belong to this category. For firm deadline or soft deadline applications, however, missing deadlines leads to a performance penalty but does not entail catastrophic results. The distinction between firm deadline and soft deadline applications lies in their view of late tasks. For firm deadline applications, completing a task after its deadline has expired is of no utility

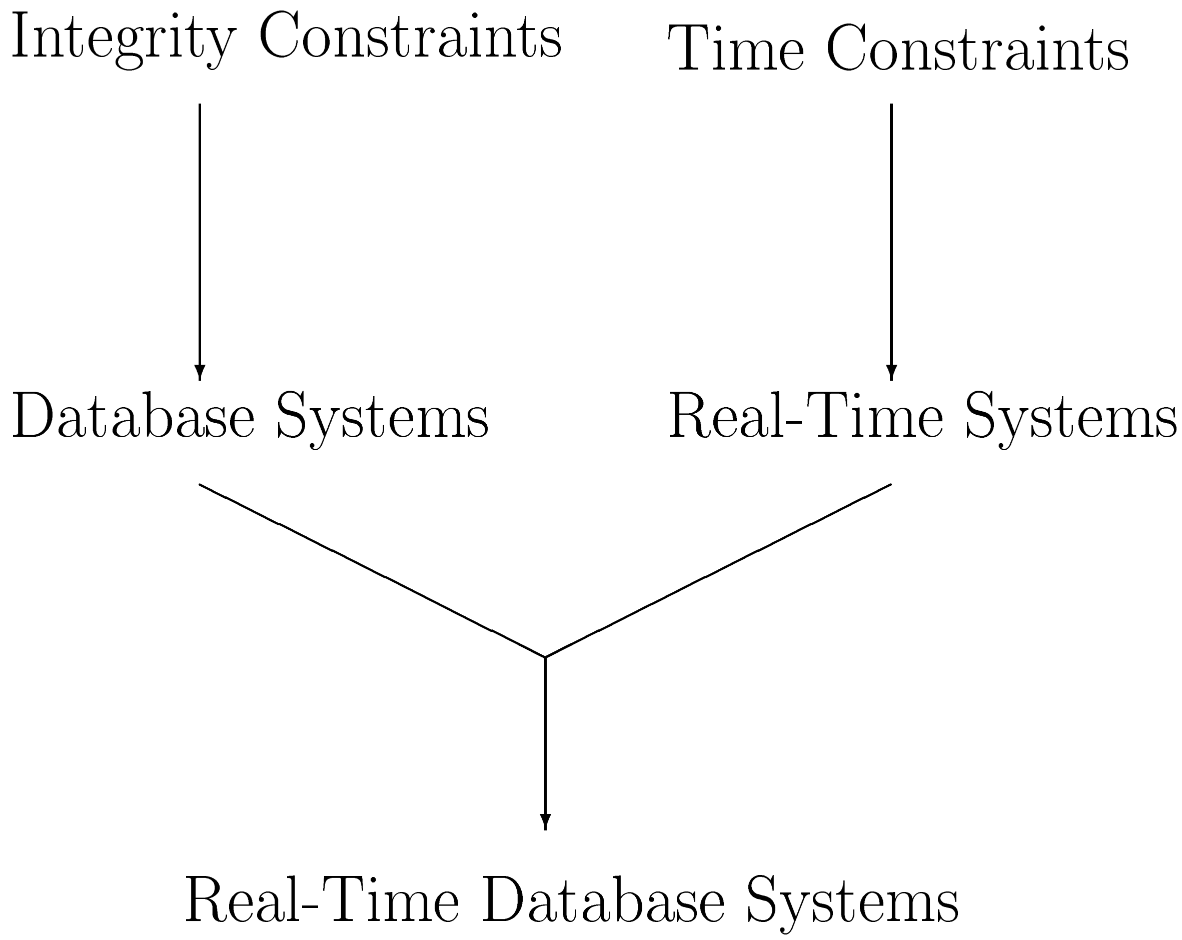

Figure 2.1: Constraints in Real-Time Database System

and in fact may be harmful. Therefore, late tasks are required to be permanently aborted and discarded as shown in Figure 2.2b. For soft deadline applications, however, there is some diminished utility to completing tasks even after their deadlines have expired (Figure 2.2c). Financial and manufacturing applications usually fall under either firm deadline or soft deadline categories.

Database systems for efficiently supporting hard deadline real-time applications, where *all* transaction deadlines have to be met, appear infeasible [24]. This is because there is usually a large variance between the average case and worst case execution times of a

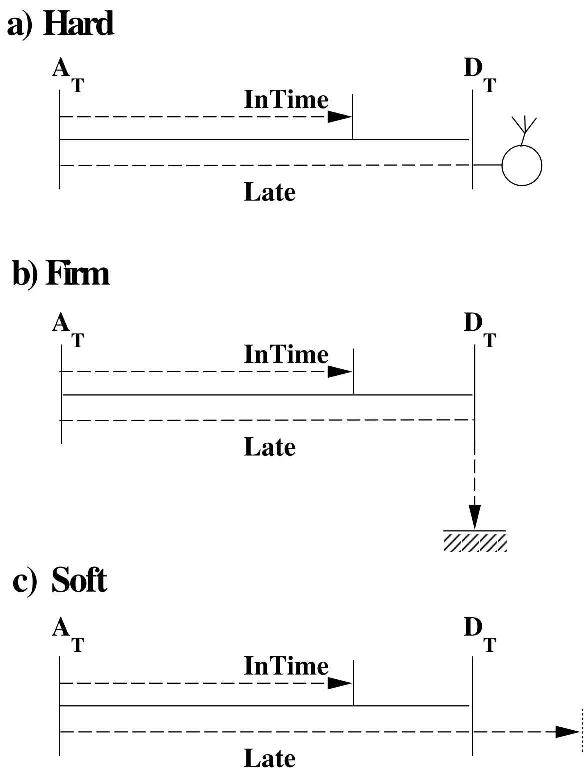

Figure 2.2: Deadline Types

transaction. The large variance is due to transactions interacting with the operating system, the I/O subsystem, and with each other in unpredictable ways. Guaranteeing transactions under such circumstances requires an enormous excess of resource capacity to account for the worst possible combination of concurrently executing transactions [24].

Database systems for firm deadline or soft deadline applications, however, appear feasible since the system is only expected to make a "best effort" to meet transaction deadlines [7]. In this dissertation, we restrict our attention to firm deadline applications, and late transactions are therefore considered to be worthless and are discarded by the database system.

2.3.1 Priority Assignment

In an RTDBS, improving the response time of one transaction at the expense of another *can* result in a system performance improvement [7]. Imagine the case where two transactions arrive at the database system, one with a tight deadline and the other with a loose deadline. If the transactions share the system resources on an equal basis, the urgent transaction misses its deadline while the slack transaction completes in time. Alternatively, if the urgent transaction progresses at the expense of the slack transaction, it completes before its deadline. The slack transaction then, by virtue of its loose deadline, completes in time despite its retarded progress. From the above discussion, it is clear that providing differential service to transactions in an RTDBS can result in a performance improvement.

The priority assignment policy decides the service precedence among transactions to the database system resources. The goal of a priority assignment policy is to improve the system performance. Popular examples of such priority assignment policies include *Earliest Deadline* and *Least Slack*. The Earliest Deadline priority mapping assigns higher priority to transactions with earlier deadlines. The Least Slack policy gives maximum priority to the transaction that has least time to finish its remaining execution. However, this priority assignment requires apriori knowledge of the transaction's execution time and obtaining such knowledge maybe difficult in a general real-time transaction processing scenario. Therefore, in our work, we use the Earliest Deadline priority assignment policy in all our experiments.

2.4 Database Security

Having discussed both basic and real-time transaction processing, we now move on to discussing database security aspects. A database system typically has *objects* and *subjects*. An object is a data item, whereas a subject is a process that requests access to an object. For example when a process requests access to a file, the process is the subject and the file is the object. A subject can access an object in different access modes (e.g., read, write, append), governed by the system security policy. Database security deals with the measures, policies and mechanisms used to provide *availability*, *secrecy* and *integrity* in a database system. The *availability* ensures that unauthorized subjects do not deny system services to genuine subjects. *Secrecy* deals with the prevention of unauthorized subjects from accessing information from or about the objects, and *integrity* deals with the prevention of the corruption of system objects either directly or indirectly by unprivileged subjects.

Ensuring security in databases translates to implementing a mechanism for monitoring and controlling the transaction data accesses. There are two types of data access control methods widely used for providing security:

Discretionary Access Control: Here the the data access control is left to the user's discretion. If a user wishes his/her data to be protected, the system offers a facility for doing so.

Mandatory Access Control: Here security is the responsibility of the system and it is required to prevent subjects (e.g., users, programs) from accessing objects (e.g., files, directories, devices, windows, sockets) that they are not trusted to handle.

The discretionary protection models are often simple and less secure whereas the mandatory protection models are more difficult to implement but guarantee certain levels of security. In this thesis, we focus on the secrecy aspects of SRTDBS that enforce Mandatory Access Control.

2.4.1 Mandatory Access Control (MAC)

Implementing mandatory protection requires assigning of sensitivity labels to all subjects and all objects in the system. Each object in the system has a *classification level* based on the security requirement. Similarly, each subject has a corresponding *clearance* level based on the degree to which it is trusted by the system. For convenience we assume, from here on, that the "level" of an object means its security classification level and the "level" of a subject means its security clearance level. Mandatory Access Controls use sensitivity labels to determine who can access what information in the system. In a *closed system* (from a security view point), only explicitly authorized object accesses are allowed, whereas in an *open system*, accesses that are not explicitly forbidden are allowed. The studies reported in this thesis were conducted on a closed system.

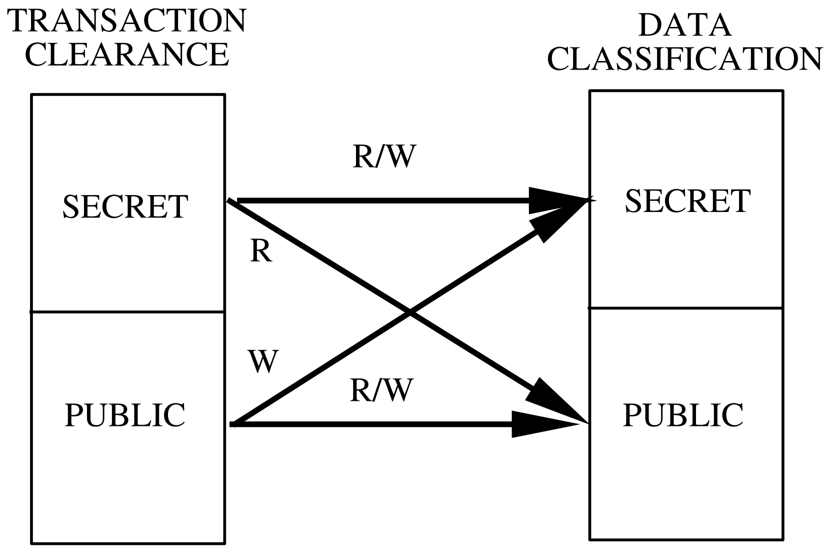

Figure 2.3: Bell-LaPadula access restrictions

2.4.2 Security Models

A security model provides a high level picture of the protection requirements in a system. In recent years, many security models have been proposed and some of these models enforce mandatory access control. In the remainder of this section, we discuss two important security models included in our studies, namely, *Bell–LaPadula* and *Restricted Write* – both these models support mandatory access control.

2.4.3 The Bell–LaPadula Model

The Bell–LaPadula model [32] imposes two restrictions on all data accesses:

- Simple security property: A subject is allowed *read* access to an object only if the former's clearance is <u>higher than or identical to</u> the latter's classification.

- Star Property: A subject is allowed *write* access to an object only if the former's clearance is <u>identical to or lower than</u> the latter's classification.

The Bell–LaPadula conditions effectively enforce a "read below, write above" constraint on transaction data accesses (an example is shown in Figure 2.3) and thereby prevent flow of information from lower security levels to higher security levels.

2.4.4 The Restricted Write Model

In the Restricted Write model, the read access condition is identical to that of the Bell–LaPadula model, but the write access condition is further restricted to the following:

- A subject is allowed *write* access to an object only if the former's clearance is <u>identical to</u> the latter's classification.

Note that the above requirement disallows *blind writes* by low security transactions to high security data are disallowed. Therefore, the model prevents lower level subjects from corrupting the secret data.

2.4.5 Covert Channels

The security models discussed above are sufficient to prevent *direct* unauthorized access to secure data. They are not sufficient, however, to protect from "covert channels". A covert channel is an *indirect* means by which a high security clearance transactions can transfer information to a low security clearance transaction [33]. There are two types of covert channels: *storage channels* and *timing channels*. Covert storage channels include all mediums that would allow the writing of a storage location by one transaction and the reading of it by another. Covert timing channels include all mediums that would allow the high security clearance transaction to signal information to another low security clearance transaction by modulating its own use of system resources in such a way that the change in response time observed by the low security clearance transaction would provide information. In this thesis, we focus our attention exclusively on <u>timing channels</u>.

We explain the timing covert channels further with the help of Table 2.1a and 2.1b, where the request/grant sequence of a Secret class transaction and a Public class transaction with regard to an exclusive resource d is shown. In Table 2.1a, where the Public class transaction requests exclusive access to the resource d at time $t = 10ms$, the request is granted immediately since no transaction is holding the resource d. In contrast, in Table 2.1b, the same request is not immediately granted since the Secret transaction is already holding the resource from $t = 5ms$ onwards. The Public transaction, in this case,

Transaction Class	Time (ms) 05	10	20
Secret			
Public		Lock(d)	

Table 2.1: a) Secret sending 0 to Public

Transaction Class	Time (ms) 05	10	20
Secret	Lock(d)		UnLock(d)
Public		LockWait(d)	Lock(d)

Table 2.1: b) Secret sending 1 to Public

is made to wait until the Secret transaction releases the resource at $t = 20ms$. The above example illustrates that the *delay* experienced by the Public transaction for accessing the data item d can be *decided* by the Secret transaction (by choosing to access or not access the data item d). The presence or absence of the delay can be used as a "signaling" or encoding mechanism by the Secret transaction to pass on information to the Public transaction – in Table 2.1, a single bit of information (*one* or a *zero*) is transmitted in this manner.

Covert channels that use the database system's *physical resources* as the medium for passing on information are relatively straightforward to tackle – for example, by introducing "noise" in the form of dummy transactions that make use of these resources. However, this approach is impractical for covert channels that use *data* as the medium (for example, presence or absence of a lock on a pre-determined data item). This is because, unlike physical resources which are typically few in number, the number of data items is usually enormous, especially in a database system. In fact, in heavily loaded systems, noise at the physical resources may be generated "for free", but this will probably never be the case for data since it is trivial to insert an additional data item that is of relevance only to the conspiring transactions. Therefore, *explicitly making data access covert-channel-free is more vital than doing the same for resource access.*

Covert channels based on data can be prevented by providing *higher* priority to the low security transaction whenever a data conflict occurs between a low security transaction and a high security transaction. Taking this approach ensures that low security transactions do not "see" high security transactions and are therefore unable to distinguish between their presence or absence. This notion is formalized in [6] as *non-interference*. From a database system perspective, it translates to implementing transaction processing components that support the non-interference feature.

Some physical resources (e.g. the buffer pool), however, require careful handling due to the multitude of covert signaling mechanisms possible in such resources (see Section 2.5.4).

2.4.6 Covert Channel Security Requirements

The necessary and sufficient conditions for a secure interference-free scheduler are given in [36], and this framework is applicable to the secure firm real-time transaction processing environment. There are three requirements to be satisfied for a system to be covert channel secure:

Value Security: A system is Value Secure if the value of objects accessed by a subject are not changed by actions of subjects belonging to higher clearance levels.

Delay Security: A system is Delay Secure if the delay experienced by a subject is not affected by actions of subjects having higher clearance levels.

Recovery Security: A system is Recovery Secure if the occurrence of a transaction restart appears the same for a subject regardless of the presence or absence of higher level subjects in the system.

2.5 Real-Time Database Security

We have discussed above, individually, database processing, real-time processing, and database security. We now shift our attention to the combination of these three areas, namely, secure real-time database systems.

Besides meeting the traditional database correctness requirements, an SRTDBS has to *simultaneously* satisfy two additional requirements, namely, provide security and minimize the number of missed transaction deadlines. Unfortunately, the mechanisms for achieving the individual goals often work at cross-purposes. In an RTDBS , high priority is usually given to transactions with earlier deadlines in order to help their timely completion. On the other hand, in secure database systems, low security transactions are given high priority in order to avoid covert channels. Now consider the situation wherein a high security process submits a transaction with a tight deadline in a SRTDBS. In this case, it becomes difficult to assign a priority since assigning a high priority may cause a security violation whereas assigning a low priority may result in a missed deadline.

2.5.1 Secure Transaction Scheduling

An SRTDBS transaction scheduling algorithm (see Figure 2.4) includes the components of Service Eligibility, Priority Assignment, Resource Service, Concurrency Control and Recovery, just as in (unsecure) RTDBS. The *Service Eligibility* component decides which transactions are allowed to execute in the database system. All other transactions are immediately (and permanently) discarded from the system. The *Priority Assignment* component assigns priorities to transactions that are eligible to execute in the system. The *Resource Service* component maps a scheduling discipline to each hardware resource. Finally, the *Concurrency Control* component maintains database consistency and resolves data conflicts. In addition to these four components, a new component that is specific to the secure environment is *Access Control* which enforces the security constraints for transaction data accesses.

A sample transaction scheduling algorithm, described in terms of the above components, is the following: { Feasible Only; Earliest Deadline; Priority PR (CPU), Priority LRU (Memory), Priority HOL (Disk); 2PL-HighPriority; Bell-LaPadula }. This algorithm implements the Feasible Only eligibility test which discards late transactions and permits only feasible transactions to execute in the system. The Earliest Deadline priority mapping assigns higher priority to transactions with earlier deadlines. The Priority

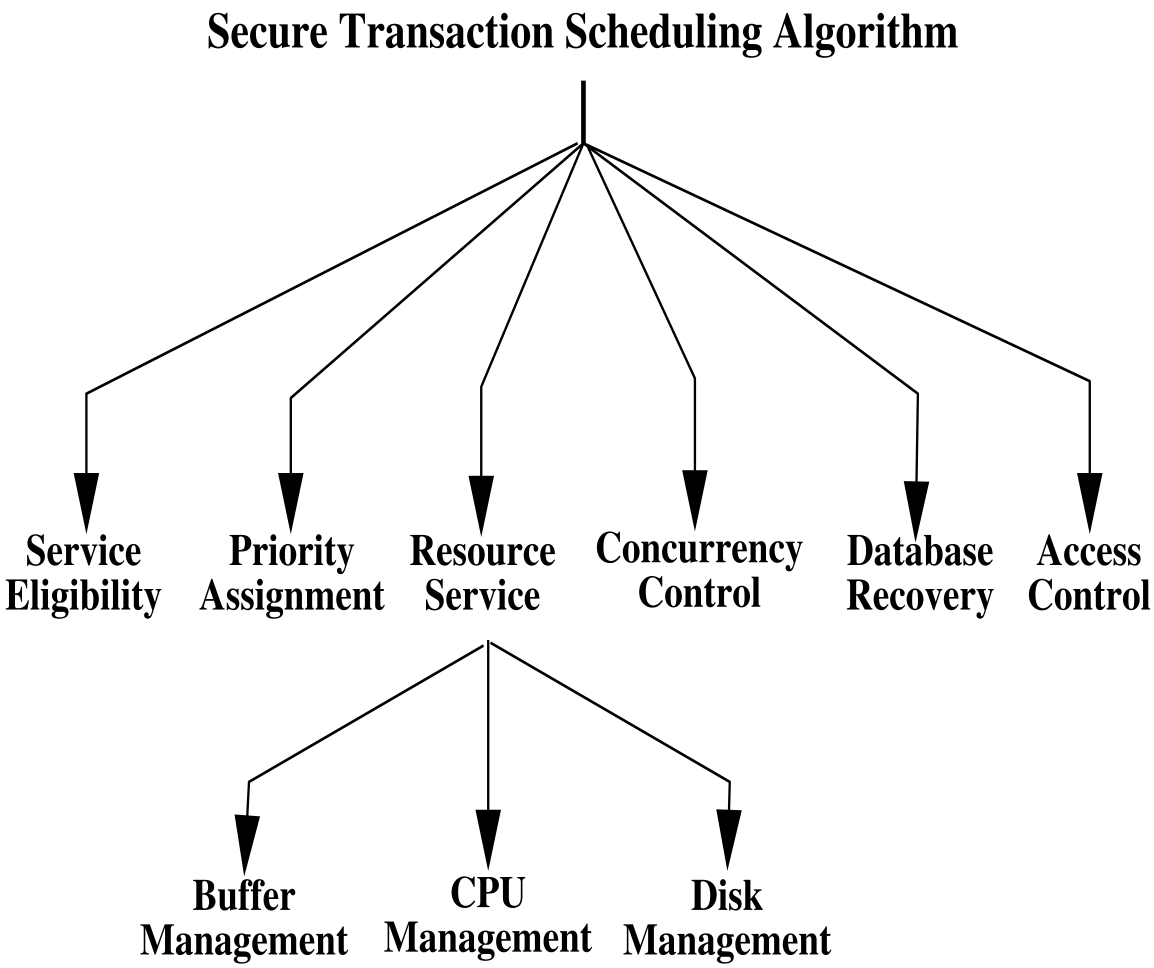

Figure 2.4: Components of Secure Transaction Scheduling Algorithm

Pre-emptive Resume [17] and Priority LRU scheduling disciplines provide priority-based preemptive service at the processors and memory buffers, respectively, while Priority HOL provides priority-based non-preemptive service at the disks. The 2PL-HighPriority CC protocol maintains database consistency with a two-phase locking scheme and resolves data conflicts in favor of higher priority transactions. Finally, the Bell–LaPadula access restrictions prevent direct unauthorized access to secret data.

Note that the above system is still vulnerable to the indirect means of information transfer through covert channels. In this thesis we explore ways to make the components of the transaction processing system covert channel secure.

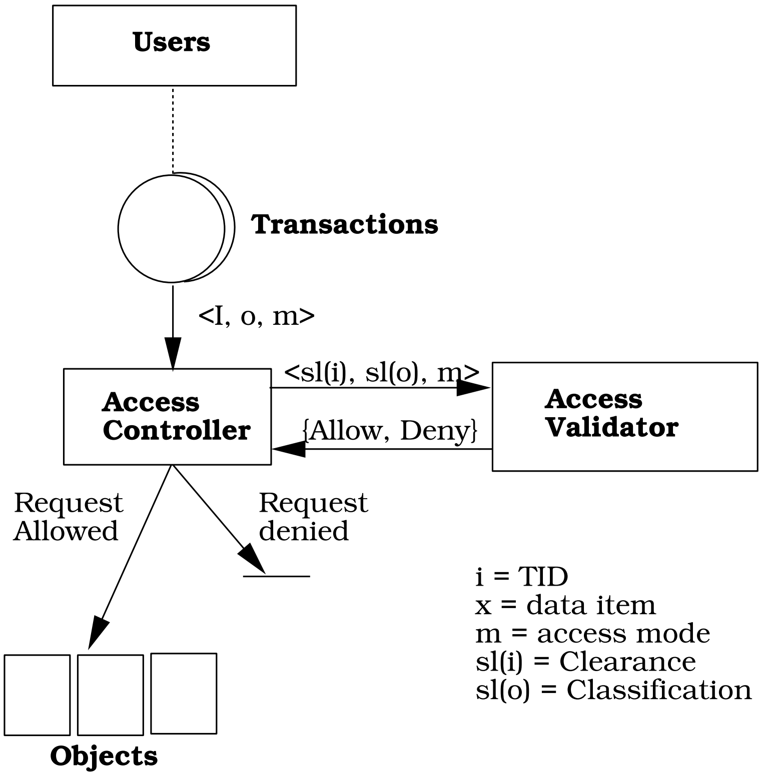

Figure 2.5: Access Control in Databases

Priority Assignment

The priority assignment in a secure RTDBS should be such that the low security level transactions should not be able to feel the presence of higher security level transactions in the system. Therefore, whenever there is a data or resource conflict between a high security level transaction and a low security level transaction, the low security level transaction should be given priority over the high security level transaction. Note that with a secure priority assignment policy in place, security violations through covert channels are possible only at non-pre-emptable non-sharable resources.

2.5.2 Access Control

For security reasons, every transaction data access request must be validated and permitted by the access control unit. Upon receiving a validation request, the access control algorithm compares the security classification level of the transaction and the security clearance level of the object to which the access is requested. If the requested operation could be permitted as per the security policy, the request is allowed. On the other hand, if a request fails in the validation process, the requester (transaction) is killed and discarded from the system. Figure 2.5 shows the access control unit in a secure transaction processing system. The request received from the transaction is of the form $< I, o, m >$, where m is the operation the transaction I wishes to perform on the object o. The access controller consults the authorization database (not shown in the figure) and obtains the security clearance level of the transaction I (indicated as $sl(I)$) and the security classification level of the object o (indicted as $sl(o)$). The tuple $< sl(I), sl(o), m >$ is passed to the access validator for validation. Depending on the return value of the access validator, the access controller either allows or denies the request.

2.5.3 Concurrency Control

In an RTDBS, the decision on which transaction's progress is to be hindered is usually based on the transaction priority. A high priority transaction should not be either restarted or blocked by a transaction with a low priority. In conjunction with a **secure** priority assignment, it would seem at first glance that, in principle, any real-time CC protocol could be used in a secure RTDBS and that the actual choice of protocol would be based only on the relative performance of these protocols. However, *not all the previously proposed real-time CC algorithms are amenable to supporting security requirements*, because a real-time CC protocol that permits, to even a limited extent, high priority transactions to be adversely affected by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature, cannot be used in a secure RTDBS.

2.5.4 Resource Service

The main physical resources in a database management system are the processors, the memory buffers, and the disks. In a secure RTDBS, service should be provided in a manner that reflects transaction priorities as well as security objectives. One approach is to make the resources behave like *priority preemptive resume* servers. In such servers, the highest-priority requester is always the one receiving service. This is achieved by preempting lower priority requesters from use of the resource. Hardware restrictions, however, prevent certain actions such as disk service, from being preemptable. This may open up covert channel possibilities and for fully addressing this issue, special hardware solutions may have to be developed.

2.5.5 Database Recovery

The role of the secure recovery mechanism is to ensure transaction atomicity and durability without opening up the possibility of covert channels. Secure recovery has received less attention from researchers than concurrency control [2]. The main reason for this is that most recovery algorithms do not require trust (i.e., privilege). It is possible to maintain a separate recovery log for each active security level and the recovery at each level can proceed independently. Therefore, the interference channels are easily eliminated.

However, ensuring the security of recovery protocols in *distributed* database systems is more complex as compared to centralized database systems. In distributed systems, even after a site decides to locally commit a cohort (the fragment of the global transaction executed at that site), that is, after it has entered the so-called *prepared state*, the local transaction manager has to retain some or all of its locks until the global decision on the transaction is available – this retention is fundamentally necessary to provide atomicity. In such a scenario, if a cohort requests a conflicting lock on the data element locked by a higher security level prepared cohort, the requester *has to* wait till the lock holder aborts or commits. However, note that this mechanism could be exploited by malicious high priority transactions as a signaling mechanism. Therefore, special mechanisms are required to ensure security of distributed commit protocols.

2.6 Secure RTDBS Architecture

A high-level "black-box" view of a SRTDBS is shown in Figure 2.6. In this model, the input process to the SRTDBS is an arrival stream of transactions with each transaction

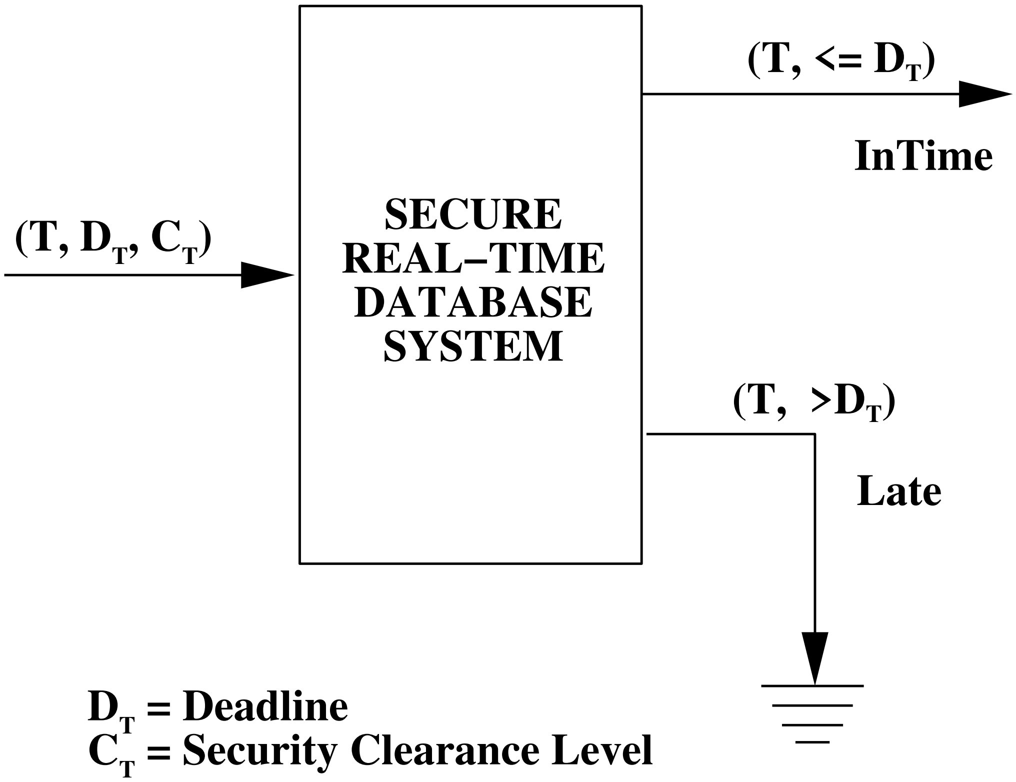

Figure 2.6: Black-Box View of the secure RTDBS

T having an associated arrival time A_T, deadline D_T, and security clearance level C_T. There are two transaction output streams from the secure RTDBS, *InTime* and *Late*. Transactions that complete at or before their deadlines join the InTime stream, while transactions that miss their deadlines join the Late stream. Transaction scheduling within the SRTDBS incorporates the architecture shown in Figure 2.7. The *eligibility monitor* tracks the eligibility status of all transactions executing in the system. A transaction that becomes ineligible is immediately removed from the system and discarded. The *priority mapper* unit assigns a priority vector P to each transaction on its arrival, and this priority is used by the hardware resource schedulers and the CC mechanism to distinguish among transactions. As mentioned earlier, we restrict our attention in this study to applications that have firm deadlines. This means that late transactions are immediately discarded and do not execute to completion. A second restriction is that the SRTDBS has no a-priori knowledge of either transaction processing requirements or transaction data accesses. This means that, from the secure RTDBS perspective, transactions are distinguished only by

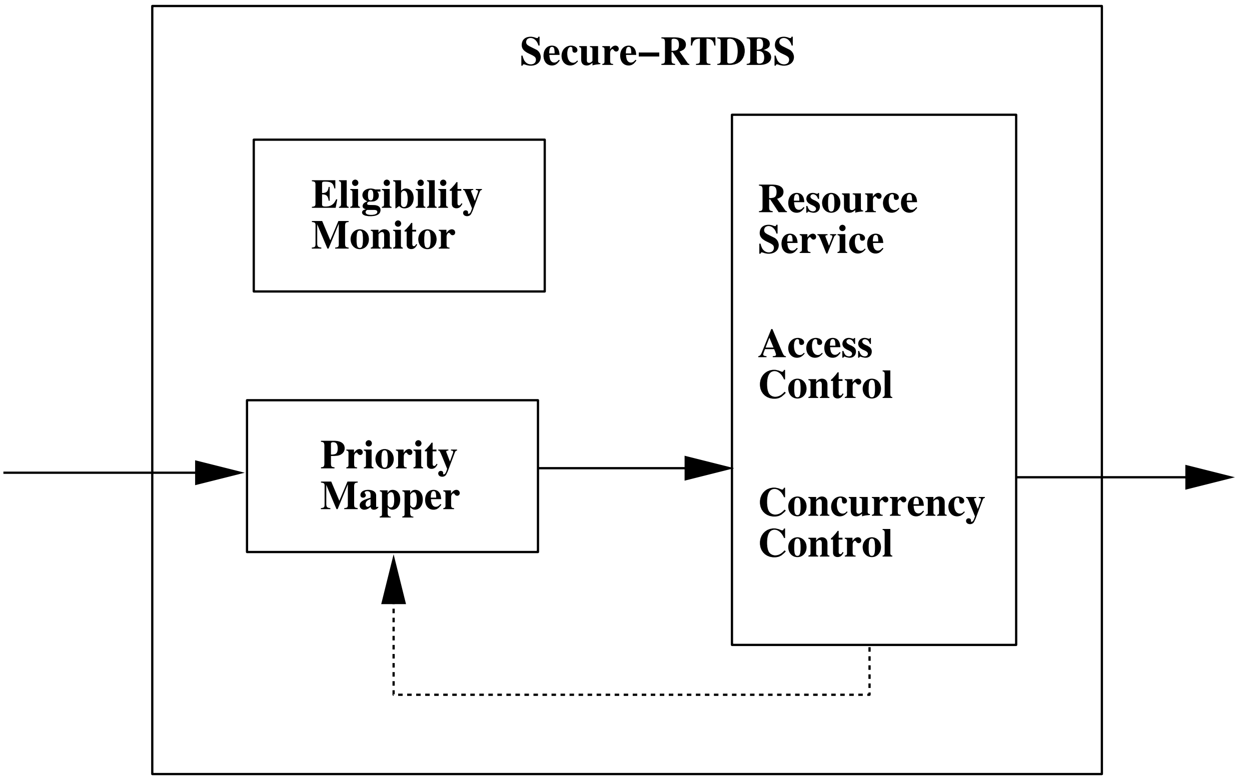

Figure 2.7: Transaction Scheduling Architecture

their arrival time, deadline and security classification level characteristics. It also means that transactions are detected to be late only when they *actually* miss their deadline since the secure RTDBS cannot estimate the remaining service requirements of a transaction (i.e. transactions remain eligible until their deadlines expire).

CHAPTER

3

Distributed Commit Protocols

A distributed transaction executes at multiple sites. To ensure the atomicity of the distributed transaction, all cohorts of the transaction must reach a uniform decision (commit or abort). This guarantee is provided by the *transaction commit protocols*. In this chapter, we will discuss the classical two-phase commit protocol, its variations such as presumed abort and presumed commit protocols, and the three-phase commit protocol. Then, we will describe a new protocol, called OPT, that allows the transactions to access uncommitted data—in a controlled fashion—in order to improve the performance of the system.**Material for this chapter is taken from [30]**

3.1 Recovery Protocols for Distributed Databases

The recovery manager provides the semantics of atomicity and durability to the transactions. *All* of the effects of the committed transactions and *none* of the effects of the aborted transactions must be visible to the database. This must be guaranteed even in the event of failures. After the system recovers from a failure, it has to be brought to a consistent state—a state where all the effects of the transactions that had committed before the system crash, and none of the effects of the transactions that were either aborted before the crash or executing at the time of the crash, are visible to the system.

The updates of some transactions that were executing at the time of crash could have reached the disk (stable storage) before the crash occurred. Atomicity requires these effects to be *undone* when the system recovers from the failure, because these transactions were not committed by the time crash occurred. Also, the updates of some transactions that had committed just before the crash might not have reached the disk—these were lost with the volatile storage as it was wiped out in the crash. These actions have to be *redone* after the system recovers from the crash to provide the durability of the committed transactions. Therefore, the system must have enough information on the stable storage before the crash occurs in order to be capable of carrying out these undo/redo operations. This simply means that the system must have the necessary information on the stable storage *at all times*—the crashes do not tell about their intentions beforehand!

Write-Ahead Logging and Commit Point

In order to ensure that the system has the undo/redo information (information required to undo/redo the actions of a transaction) in the stable storage *at all times*, the recovery manager uses a technique called *write-ahead logging (WAL)*. Before a data item is modified by a transaction, a log record is generated having sufficient information to undo or redo the effect of the modification. The system ensures that the log record, and not the updated data item, is the first to reach the stable storage. This is ensured by using *Log Sequence Numbers (LSN)*. An efficient technique called ARIES that uses WAL and LSN mechanisms is described in [41]. Thus, the WAL and the LSN mechanisms ensure that there is always sufficient information on the stable storage to undo/redo the effects of a transaction. Usually, these undo/redo log records are *idempotent*—their repeated application to a transaction is same as applying them only once. This becomes important, for example, when a site fails during the recovery process itself. When the site comes up, the recovery process might re-apply some of the redo/undo records that had been applied previously.

After a transaction has completed its execution and is ready to commit, a `COMMIT` log record is generated. This log record is *forced* to the stable storage, that is, written in a synchronous fashion (the transaction waits until this record reaches the disk). The LSN mechanism mentioned above ensures that all the WAL records belonging to the transaction reach the stable storage before this commit log record. Thus, when the commit log record reaches the disk, there is sufficient information on the disk to redo the effects of the transaction in case a failure occurs. The point at which the commit log record reaches the disk is called the `commit point` and the transaction is said to be *committed* at this point.

We see that the writing of the log record to the disk can be done in two ways: (1) synchronously, where the execution of the transaction is suspended until the writing of the record is over, and (2) asynchronously, where the execution of the transaction continues normally without waiting for the writing operation to be complete. Hereafter we will refer to the latter as simply a *write* and the former as a *force-write*. Note that the force-writes have a larger performance penalty than writes because they increase the response time of a transaction.

Recovery and Buffer Management

The choice of recovery mechanisms to be used when a site recovers from a failure enforces certain constraints on buffer management, directing buffer manager to force certain pages to the disk at particular times, and not allowing it to write certain pages to the disk at some other times. This issue is discussed in elaborate detail in [42, 43]. A page modified by a transaction that has not yet committed is a *dirty* page until either the commit or the abort of the transaction is completed. Based on how the buffer manager handles the dirty and clean pages the buffer management policies can be classified as follows:

Steal/No-Steal Policy :

If the buffer manager is *not* required to distinguish the dirty pages from the clean pages when deciding which page to remove from the bufferpool, a dirty page can be written (stolen) from the buffer to the disk. This policy is called as *steal* policy. As the disk might contain dirty pages, the log must contain sufficient information for undoing the transaction effects if required. On the other hand, if the buffer manager uses a *no-steal* policy, the dirty pages are not allowed to be written to the disk and are retained in the bufferpool until the outcome of the transaction is determined. In this case, because the disk never contains the dirty pages, the log need not contain any information for the undo purposes. In case the transaction aborts, these buffer pages are simply marked as invalid.[1] However, the no-steal policy has a few disadvantages: it enforces the lock granularity to be page which is a bad idea, and it needs a prohibitively large bufferpool for long (batch) transactions.

Force/No-Force Policy :

If the buffer manager force-writes the pages modified by a transaction when the transaction is ready to commit, the policy is called as *force* policy. On the other hand if the buffer manager does not force-write the modified pages when the transaction commits, but replaces them to the disk only when required by the buffer replacement policy, the policy is called as *no-force* policy. The force policy has the advantage that the log need not contain any information for the redo purposes while such information is mandatory if the no-force policy is used. However, the force policy may require a frequently accessed page to be force-written to the disk many more times than the no-force policy.

In general, the steal/no-force combination seems the appropriate choice for practical systems [43].

The other mechanisms proposed in the literature for recovery in centralized systems include shadow paging [38], and the differential files [44]. However, due to some inherent problems in these techniques, commercial systems usually avoid them and instead prefer to implement the logging mechanisms [43]. A study of the performance issues that arise in the integration of various concurrency control protocols with the recovery protocols is given in [40].

[1]This policy, however, requires logging of undo information for on-line undo purposes (transaction aborts during normal operation). This undo log information need not be written to the disk. To avoid undo logging completely, a page modified by a committed transaction must be written to the disk before another transaction can dirty it [37].

3.1.1 Distributed Transaction Execution

A common model of distributed transaction execution is shown in Figure 3.1. At the site where the transaction is submitted, a *master* process is created to coordinate the execution of the transaction. At a site where the transaction needs to access data, a *cohort* process on behalf of the transaction is created. Usually, there is only one cohort on behalf of the transaction at each such site. The master sends the STARTWORK message to a cohort when some data at the site of the cohort is required to be accessed.[2] Depending on the transaction architecture, the master may send the STARTWORK messages to multiple cohorts without waiting for their responses, that is, the cohorts of the transaction will execute in a *parallel* fashion. Or the master may send the subsequent STARTWORK message only after the previous cohort has completed the work assigned to it, that is, the cohorts of the transaction execute one after the other in a *sequential* fashion. A cohort, after successfully executing the master's request, sends a WORKDONE message to the master. After receiving the WORKDONE message, the master may send more work to the same cohort, or may decide to send the next piece of work to another cohort of the transaction.

The master may decide to conclude the transaction when all the work assigned to the transaction is completed. Or the decision may be necessitated at the instance of the user submitting the transaction. In any case, the decision to commit or abort the transaction must be uniform—the master and all cohorts must agree on a common decision. It might seem that the fact that a cohort had sent a WORKDONE message means that the cohort was willing to commit the transaction. The problems here are these: First, the sending of the WORKDONE message does not enforce any binding on the cohort to agree on the decision. Due to reasons such as concurrency control, performance, etc. the cohort could still be aborted even after the WORKDONE message was sent. Second, there can be failures—communication links may fail, or some of the sites hosting the cohorts of the

[2]Actually, the master sends the STARTWORK message to the transaction manager of the site. The transaction manager creates a cohort on behalf of the master, if one is not already existing. As these implementation details are not important for the understanding of the concepts presented here, we will ignore them for the sake of simplicity.

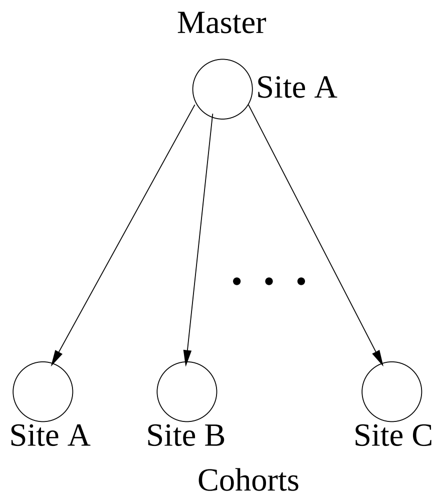

Figure 3.1: Distributed Transaction Execution

transaction may fail. Therefore, a commit protocol is needed to ensure that all cohorts and the master reach a uniform decision—a decision that will be binding on all sites even if a failure occurs.

A variety of transaction commit protocols have been devised, most of which are based on the classical **two-phase commit (2PC)** protocol [39]. The two-phase commit protocol is discussed in considerable detail in the next section.

A Digression about Failures

Two points are worth mentioning before we start the description of the 2PC protocol: First, how does a site know that a communication link or a remote site has failed? And second, how are failures handled? Generally, a timeout mechanism is used to address the first question. If the response to a message is not received from the remote site within the timeout period, the local site assumes that either the communication link to the remote site, or the remote site itself has failed.[3]. For the second question, when a site recovers from a failure, it is handed over to the recovery manager of the site. The recovery manager scans the log, and if it finds a `COMMIT` log record for a transaction, it

[3]Both site failures and the communication link failures manifest themselves in the loss of communication between the local and remote site. There is no way for the local site to tell whether it is a communication link failure, or the remote site has failed.

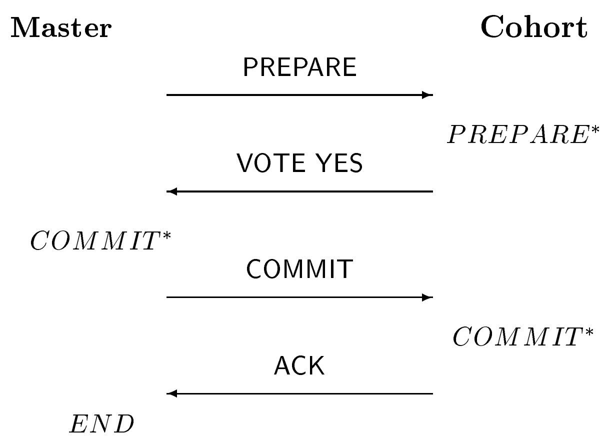

Figure 3.2: Two-Phase Commit Protocol for Committing Transactions

knows that the transaction had committed before the failure occurred. In the same way, if it finds an `ABORT` log record, it knows that the transaction was aborted before the failure occurred. In these cases, the recovery manager ensures that the effects of the committed transactions are redone and the effects of the aborted transactions are undone as was explained in Section 3.1. The only problem is for the transactions about which neither a `COMMIT` nor an `ABORT` log record is found. The actions that recovery process need to take in this case depend on the commit protocol being used—we will discuss them with the description of the various commit protocols.

3.2 Two-Phase Commit Protocol

The two-phase commit protocol operates in two phases—as the name suggests. In the first phase, the master reaches a global decision (*commit* or *abort*) based on the local decisions of the cohorts. In the second phase, this decision is conveyed to the cohorts and implemented.

For sake of clarity, we first describe the protocol without considering failures. The first phase starts when the master decides to conclude the transaction. At this point, the master sends PREPARE messages in parallel to all its cohorts. The sending of the PREPARE message is an indication to the cohorts that the master would like to commit

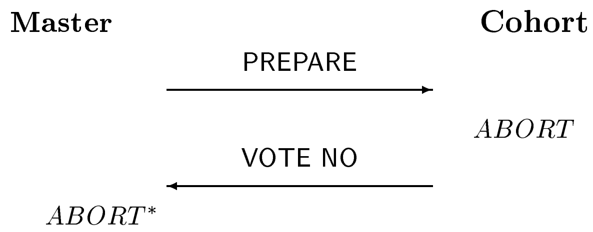

Figure 3.3: Two-Phase Commit when Cohort Votes NO

the transaction. On receiving the PREPARE message, a cohort willing to commit the transaction force-writes a PREPARE log record to its stable storage, and sends a YES vote to the master (as shown in Figure 3.2, the names on the arrows in the figure indicate the messages and the names in italics indicate the log records, an * with the name indicating that the log record is force-written). At this point, the cohort is said to be in prepared state wherein it cannot *unilaterally* commit or abort the transaction but has to wait for the final decision from the master. On the other hand, a cohort that decides to abort the transaction writes (note, not force-writes) an ABORT log record and sends a NO vote to the master (as shown in Figure 3.3). Since a NO vote acts like a veto, the cohort is permitted to *unilaterally* abort the transaction without waiting for the final decision from the master.

After the master receives the votes from all the cohorts, it initiates the second phase of the protocol. If all the votes are YES, it moves to a committing state by force-writing a COMMIT log record and sending COMMIT messages to all the cohorts. Each cohort after receiving a COMMIT message moves to the committing state, force-writes a COMMIT log record, and sends an ACK message to the master. The master writes an END log record after receiving the ACKs from all the cohorts (Figure 3.2), and then forgets the transaction, that is, the control information about the transaction is removed from memory at the master's site.

If the master receives even one NO vote, it moves to the aborting state by force-writing an ABORT log record and sends ABORT messages to those cohorts that are in the prepared state. These cohorts, after receiving the ABORT message, move to the aborting state,

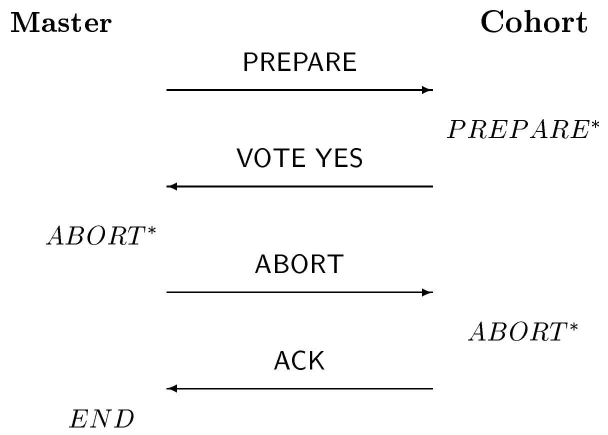

Figure 3.4: Two-Phase Commit when Cohort Votes YES but Master Decides Abort

force-write an ABORT log record and send an ACK message to the master. The master writes and END log record after receiving the ACKs from those cohorts that were sent the ABORT message, and then forgets the transaction (Figure 3.4).

If a timeout occurs while the master is waiting for the vote from a cohort, it assumes a NO vote and aborts the transaction as described above (note that the ABORT message is sent to this cohort also to ensure that it is notified about the final decision about the transaction). If a cohort's site notices before receiving the PREPARE message from the master that the master's site has failed, the cohort is simply aborted. However, if the master fails after the cohort has dispatched the YES vote, the cohort will periodically try to contact the master until a decision from the master is received. In the same fashion, if master does not receive an ACK for the decision it has sent to the cohort, it will periodically re-send the decision to the cohort until it finally receives the ACK.

3.2.1 State Transitions

The master and the cohorts visit various states during the lifetime of a transaction. Let us assume, without loss of generality, that both the master and the cohorts are in an executing state before the two-phase commit protocol is initiated by the master. From this point, a cohort can move to either aborting state or prepared state depending upon

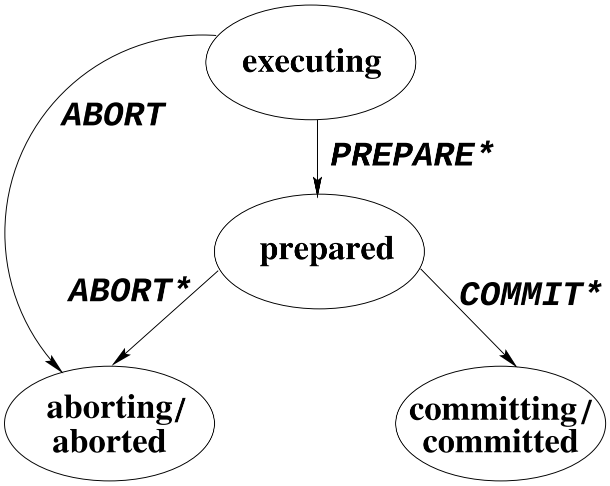

Figure 3.5: State Transitions in 2PC for the Cohort

whether it votes YES or NO to the master's request (as shown in Figure 3.5, the names on the transition arcs in the figure indicate the log records, an * with the name indicating that the log record is force-written). If it decides to vote YES, it force-writes a PREPARE log record before sending the message. The force-writing of the PREPARE record takes the cohort to the prepared state. Once a cohort is in prepared state, it can no longer *unilaterally* decide to move to the aborting state—it has to wait for the master's decision of commit or abort, based on which it will move to aborting or committing state by force-writing the appropriate log record.

The master makes the decision (commit or abort) about the fate of the transaction. Its state transitions are shown in Figure 3.6. It can move to the aborting state from the executing state without even sending PREPARE messages to the cohorts (for example, if the transaction is aborted by the user). It moves to the prepared state by sending the PREPARE messages to the cohorts. Note that this state transition does not require writing of any log record. From the prepared state, the master moves to the committing state (by force-writing a COMMIT log record) only if it gets all YES votes from the cohorts. Even if one NO vote is received, the master moves to the aborting state (by force-writing an ABORT log record). From these committing or aborting states, the master moves to the committed or the aborted states, respectively, by writing an END log record. The END log record is written only after the master has received all ACKs from the cohorts that were sent the

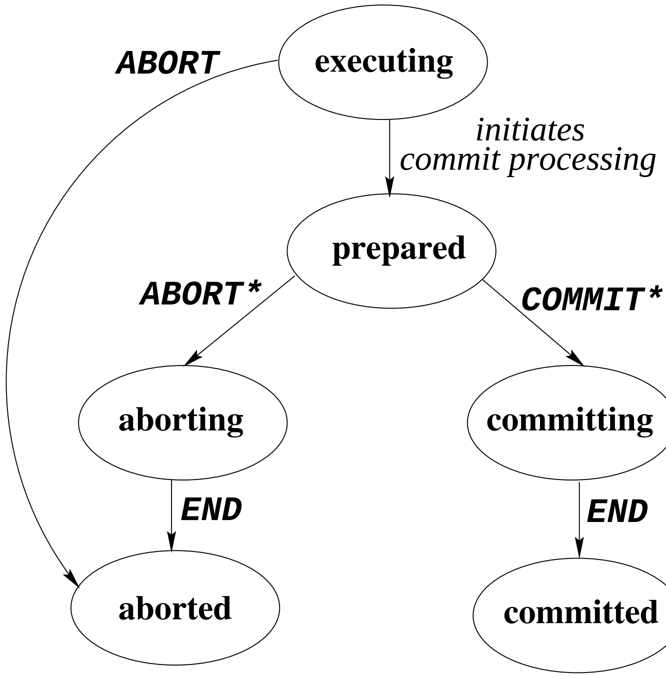

Figure 3.6: State Transitions in 2PC for the Master

decision. Note that the cohorts do not write any more log records (that is, no END record) after moving to committing or aborting states, therefore, for the cohorts, these states are equivalent to committed or aborted states, respectively.

Some of the states are *volatile*, for example, the executing state. The information about these states will not survive a system crash, and when the failed site recovers the transaction cannot resume these states. Some states are *persistent*, that is, they survive the system crash. This is because the log records before moving to these states are force-written. When the failed site recovers, the transaction can be brought back to these states by using the information available in the log. These states are prepared, committing, and aborting for the cohort, and committing and aborting for the master. The persistence of these states is required so that the recovery process can take a correct decision to ensure the atomicity of the transaction (recall from Section 3.1.1 that a failed site is handed over to the recovery process once the failure has been repaired). Finally, the durable states (committed, aborted) are the final states for a transaction. The durable states need not survive the crash but it must be possible to reach the same durable state of a transaction when a site recovers from the failure. This is fundamentally necessary to guarantee the durability of the transactions.

3.2.2 Forgetting the Transaction

When the master or a cohort of a transaction is created at a site, some data structures are allocated and control information is generated for the transaction. It is desirable that the transaction manager at the site hosting the master or the cohort (or both) *forgets* the transaction at the earliest possible time. A transaction manager forgets the transaction by releasing the data structures allocated to the transaction and by removing the control information about the transaction from memory.

Once a transaction manager forgets the transaction, it is no more in a position to answer queries about the transaction. Any further queries about the transaction will always get a `no information` response from the transaction manager. Therefore, a transaction manager forgets the transaction only when it is sure that no more queries about the transaction will be asked. Note that the `no information` response does not mean that the transaction manager has no information at all about the transaction. It simply means that no such information is available in the memory—there may be information in the log on the stable storage, but it will be too inconvenient for the transaction manager to find the information from the log in the absence of any control information about the transaction. Of course, if a crash occurs the log will be scanned while recovering from the crash and the information found there will be used to undo/redo the operations of the transactions.

The transaction manager forgets the transaction by writing an `END` log record (not forced). For a cohort, the `END` log record is merged with the `COMMIT` or the `ABORT` record itself, because the cohort can forget the transaction at the time of writing the `COMMIT` or the `ABORT` record. Thus there is no need for the cohort to write a separate `END` log record. The master, however, writes the `END` log record only when it has received the ACKs from all the cohorts that were sent the decision message. This is because if an ACK from a cohort has not been received, there is a chance that the cohort will query the master in near future about the outcome of the transaction.

3.2.3 Recovery from Failures

Let us now consider site and communication link failures. As mentioned earlier (Section 3.1.1), when a site recovers from a crash, it is handed over to the recovery manager. Recovery manager performs redo or undo operations for the transactions for which it finds `COMMIT` or `ABORT` records, respectively, in the log. If an `END` record is also found in the log for such a transaction, the recovery manager forgets the transaction. It was mentioned above that the `END` log record for a cohort is merged with the `COMMIT` or `ABORT` record. Therefore, if the recovery process finds a `PREPARE` record and a `COMMIT` or `ABORT` record for the transaction, it knows that the site was hosting a cohort of the transaction; performs the appropriate redo/undo operations; and forgets the transaction. For the master, the `END` record will be found only if all the ACKs from the cohorts were received and the log record had reached the stable storage before the crash occurred—the `END` record is not force-written to the disk. Therefore, if a `COMMIT` or `ABORT` record is found but the `PREPARE` record and the `END` record are not found in the log, this implies that: (a) the site was acting as a master for the transaction, (b) the decision about the transaction had been taken, and (c) some cohorts might not have yet received the decision. Therefore, the recovery manager periodically sends the decision message to the cohorts (the identities of the cohorts may be found in the decision record itself) until it receives ACKs from them. After receiving all ACKs, the recovery manager writes an `END` record in the log, and forgets the transaction.

For the transactions for which no log record indicating the commit processing (`PREPARE, COMMIT, ABORT`) is found, the recovery manager does not need to know whether the site was acting as the master or a cohort. It assumes that all such processes were in the executing state, and aborts them (by undoing their operations in case the undo records put by the WAL mechanism are found in the log). Note, however, that the cohorts may actually have been in the aborting state at the time of crash. But this does not affect the correctness because the recovery manager still moves the cohorts into the aborting state, and the undo operations anyway are *idempotent*.

The only case remaining is that the recovery manager finds a `PREPARE` record in the log, but finds no further COMMIT or ABORT record. This indicates that the site was hosting a cohort and the crash occurred after the cohort had sent a YES vote but before it could receive the decision from the master. On recovery from a failure, such cohorts cannot take a unilateral decision about the fate of the transaction. Therefore, the recovery manager queries the master about the outcome of the transaction (assuming that the master is up). Obviously, the master would not have received an ACK from such a cohort, therefore, the master would have retained the information about the transaction, and will reply accordingly. The recovery manager at the cohort's site will take the appropriate action based on the master's reply.

Failure-Blocking of the Cohorts

Now, consider a situation where master has dispatched the PREPARE messages but has failed before it could dispatch the decision to the cohorts. All those cohorts that had voted YES will be waiting in the prepared state for the master's decision. We call such cohorts as failure-blocked cohorts because they are waiting for the decision from the master that has failed. This is in contrast with the data-blocked cohorts where the cohorts are blocked waiting for the data items held by other cohorts. The failure-blocked cohorts will remain so until the master recovers from the failure. Moreover, these cohorts will continue to hold system resources such as locks on data items, making these unavailable to other transactions, which in turn may become data-blocked waiting for the data locks to be relinquished, i.e., "cascading blocking" results. It is easy to see that if the down-time for the master is long, it may result in major disruption of transaction processing activity in the distributed system.

No Information Case

We see till now that 2PC eliminates the possibility of the no information case which was discussed in Section 3.2.2. That is, the master site does not forget a transaction unless it is sure that no cohort will be asking it about the outcome of the transaction. Now, consider a case where the master fails after sending the PREPARE messages. A cohort on receiving the PREPARE message goes into the prepared state and sends a YES vote to the failed master. On recovery from failure, the recovery process at the master does not find any information pertaining to the commit processing of the transaction and simply aborts it and forgets it, deallocates all resources that were allocated to the transaction, and removes all information pertaining to the transaction from the memory. The cohort, on the other hand, queries the master about the outcome of the transaction. As the master has forgotten the transaction, it has no information about the transaction! Fortunately, such a case arises only in the situation discussed here. It is clear that in such a situation, the master had aborted the transaction. Therefore, a cohort on getting a `no information` response from the master can *correctly assume* that the transaction was aborted. The other solution to this situation could have been to force-write a log record at the master before sending the PREPARE message. But, the extra force-write may have an adverse impact on the performance of the commit protocol.

3.2.4 Summary of Two-Phase Commit Protocol

2PC guarantees the uniform commitment of the distributed transaction by means of messages and logging. In the absence of failures, the protocol is straightforward in that a commit decision is reached if all cohorts are ready to commit, otherwise an abort decision is taken. To handle the failures, 2PC ensures that the sufficient information is force-written on the stable storage to reach a consistent global decision about the transaction. A cohort may query the master to find out the outcome of the transaction. The 2PC ensures that the master will have the information available to answer such queries because otherwise it could lead to non-uniform commitment of the transaction (this situation, however, does occur in one special case—called the `no information` case—where the master does not have any information about the transaction to answer a cohort's query, as discussed earlier).

The message and the logging overheads in 2PC are listed in Tables 3.1 and 3.2, respectively. For each committing transaction the master sends two messages to each cohort and writes two log records one of which is forced. Each cohort sends two messages to the

Table 3.1: Message Overheads for 2PC

	Master to Cohort	Cohort to Master
For Committing Transactions	2	2
Cohort Votes No	1	1
Cohort Votes YES but Master Decides Abort	2	2

Table 3.2: Logging Overheads for 2PC

	Master		Cohort	
	Total Log Records	Force-writes	Total Log Records	Force-writes
For Committing Transactions	2	1	2	2
Cohort Votes No	2	1	1	0
Cohort Votes YES but Master Decides Abort	2	1	2	2

master and writes two log records both of which are forced. For an aborting transaction (a transaction that aborts in the commit processing phase, for example, if master receives a NO vote), the master sends two messages to each cohort that votes YES and one message to each cohort that votes NO, and writes two log records one of which is forced. Each cohort that sends a NO vote sends only one message (the NO vote), and writes only one log record which is not forced. On the other hand, each cohort that votes YES sends two messages and writes two log records both of which are forced.

In summary, the protocol guarantees atomicity and durability of the transactions despite the occurrence of site or communication link failures. The protocol, however, can lead to `failure-blocking` of the cohorts in case of failures.

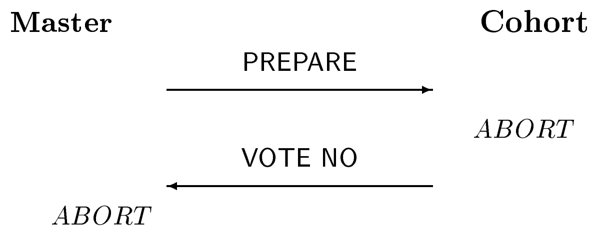

Figure 3.7: Presumed Abort when Cohort Votes NO

3.3 Presumed Abort Protocol

As described in the previous section, the 2PC protocol requires transmission of several messages and force-writing of several log records. A variant of the 2PC protocol, called **presumed abort (PA)** [35], attempts to reduce these overheads by requiring all cohorts to follow a "in the `no information` case, abort" rule. We saw in the previous section that when the `no information` case occurs, it can be *correctly* assumed that the transaction was aborted. PA attempts to save on a few messages and log writes in such a way that there will be many more situations when the `no information` case will occur. But the protocol guarantees that in all such situations, the transaction can be correctly assumed to have been aborted. In other words, if after coming up from a failure a site queries the master about the final outcome of a transaction and finds no information available with the master, the transaction is correctly assumed to have been aborted. With this assumption, it is not necessary for the cohorts to either send acknowledgments for the ABORT messages from the master or to force-write the `ABORT` log record. It is also not necessary for the master to force-write the `ABORT` log record or to write an `END` log record after the `ABORT`. Figure 3.7 shows the execution of PA for a cohort that votes NO, and Figure 3.8 shows the execution of PA when the cohort votes YES but the master decides to abort the transaction (some other cohort could have voted NO).

When a site recovers from a failure, the only case it will query the master about the outcome of the transaction is that it finds a `PREPARE` record in the log. If the master had decided the commit, then it cannot forget the transaction without receiving the ACK from the cohort. Therefore, in such a case, master will respond with the COMMIT message. On

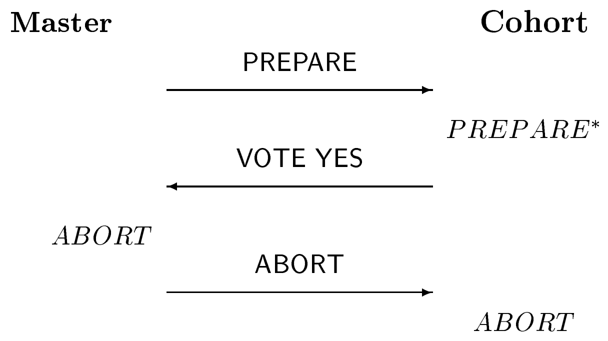

Figure 3.8: Presumed Abort when Cohort Votes YES but Master Decides Abort

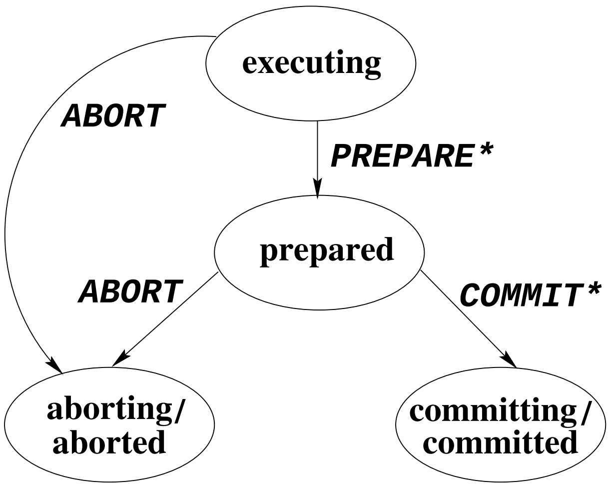

Figure 3.9: State Transitions in PA for the Cohort

the other hand, if the decision was abort, the master would have forgotten the transaction without waiting for the ACK from the cohort. Therefore, on getting a `no information` response, the cohort can correctly assume that the transaction was aborted. In all other cases, when the recovery process does not find any commit processing record in the log, it will simply (and correctly) abort the master/cohort as described in Section 3.2.3. We will not discuss about the committed transactions as they are handled in exactly the same way as 2PC. The state transition diagrams for PA are shown in Figures 3.9 and 3.10.

In short, the PA protocol behaves identically to 2PC for committing transactions, but has reduced message and logging overheads for aborted transactions.

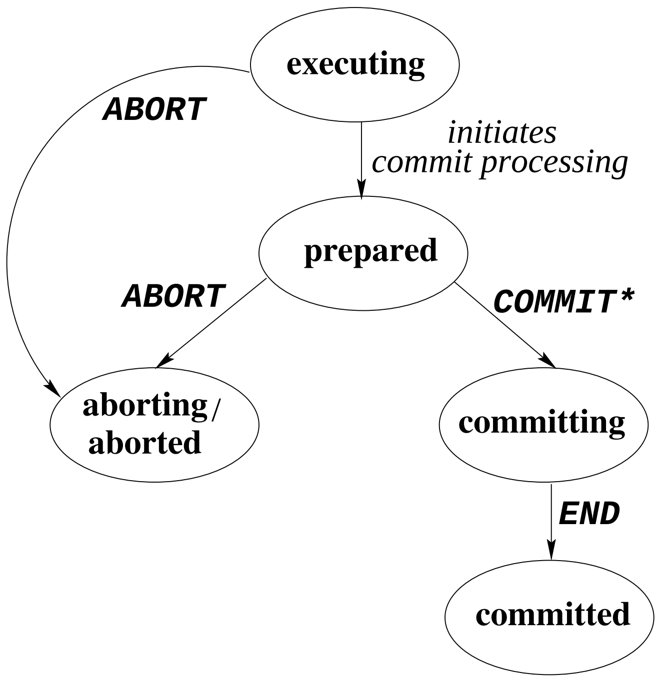

Figure 3.10: State Transitions in PA for the Master

3.4 Presumed Commit Protocol

In general, the number of committed transactions is much more than the number of aborted transactions. Based on this observation, a variation of presumed abort is proposed that attempts to reduce the messages and logging overheads for *committing* transactions rather than *aborting* transactions. This variation, called **presumed commit (PC)** [35], requires all cohorts to follow a "in the `no information` case, commit" rule. In this scheme, cohorts do not send acknowledgments for the COMMIT global decision, and do not force-write the `COMMIT` log record. In addition, the master does not write an `END` log record (master does force-write the `COMMIT` log record). Instead, the cohorts send acknowledgements for the ABORT decision, and `ABORT` log records are force-written. The result is that when the `no information` case arises, the cohort can correctly assume that the transaction was committed.

However, consider a situation where the master fails after sending the PREPARE messages. When the master recovers from the failure, the recovery process does not find any commit processing record for the transaction in its log. Therefore, the recovery process

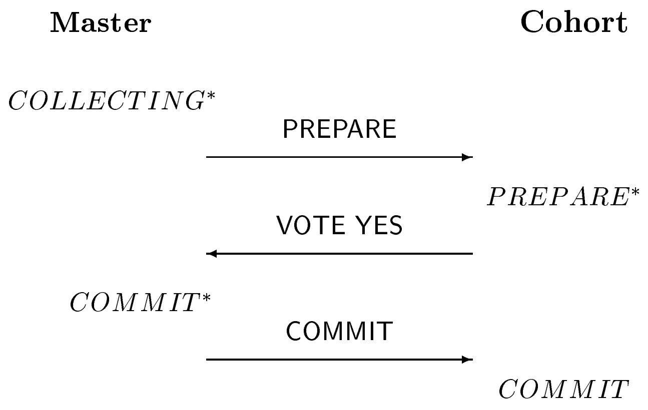

Figure 3.11: Presumed Commit for Committing Transactions

simply aborts, and *forgets* the transaction. Now, a cohort that has entered the prepared on receiving the PREPARE message queries the master about the outcome of the transaction. As the master's site has already forgotten the transaction, the response will be a `no information` case. By following the "presumed commit" rule, the cohort will decide the commit. However, the transaction had been aborted by the recovery process at the master's site. Thus, it leads to inconsistent decision about the transaction. In order to solve this problem, the master is required to force-write a `COLLECTING` log record before initiating the commit protocol. This log record contains the names of all the cohorts involved in executing that transaction. Figure 3.11 shows the execution of PC for committing transactions. The state transition diagrams for PC are shown in Figures 3.12 and 3.13.

In summary, presumed commit reduces the message and logging overheads for the committing transactions, but increases the overheads of aborting transactions. Except for the extra `COLLECTING` record, the abort processing in PC is same as in 2PC.

Significance of PA and PC

The PA and PC optimizations of 2PC have been implemented in a number of database Products. These protocols were developed and implemented in the $R^\star$ project at IBM Almaden Research Center. PA has been incorporated in many research prototypes and

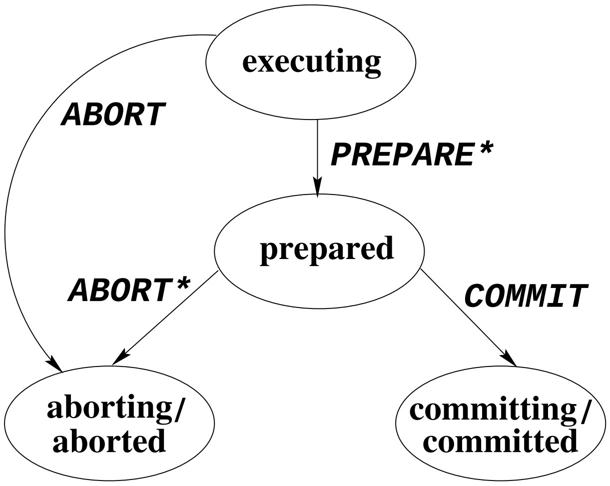

Figure 3.12: State Transitions in PC for the Cohort

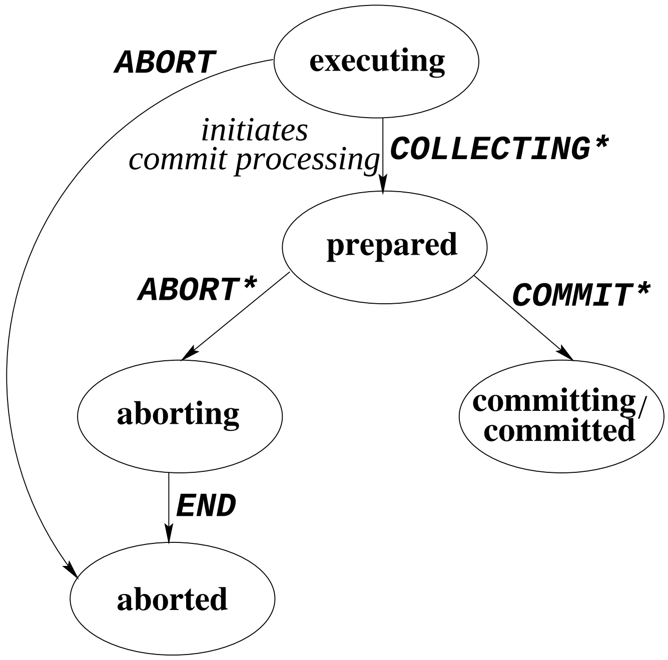

Figure 3.13: State Transitions in PC for the Master

commercial products, for example, Tandem's TMF, DEC's VAX/VMS, Transarc's Encina, CMU's Camelot, Unix System Laboratories' TUXEDO, and IBM Almaden Research Center's QuickSilver in addition to $R^\star$ [34].

3.5 Three-Phase Commit Protocol

A fundamental problem with all of the above protocols is that cohorts may become *blocked* in the event of a site failure and remain blocked until the failed site recovers (`failure-blocking` was also discussed in Section 3.2.3). For example, if the master fails after initiating the protocol but before conveying the decision to its cohorts, these cohorts will become blocked and remain so until the master recovers and informs them of the final decision. During the blocked period, the cohorts may continue to hold system resources such as locks on data items, making these unavailable to other transactions, which in turn become blocked waiting for the resources to be relinquished, i.e., "cascading blocking" results. It is easy to see that, if the blocked period is long, it may result in major disruption of transaction processing activity.

To address the `failure-blocking` problem, a **three-phase commit (3PC)** protocol was proposed . This protocol achieves a non-blocking[4] capability by inserting an extra phase, called the "precommit phase", in between the two phases of the 2PC protocol. In the precommit phase, a preliminary decision is reached regarding the fate of the transaction. The information made available to the participating sites as a result of this preliminary decision allows a global decision to be made despite a subsequent failure of the master. Note, however, that the price of gaining non-blocking functionality is an increase in the communication overheads since there is an extra round of message exchange between the master and the cohorts. In addition, both the master and the cohorts have to force-write additional log records in the precommit phase.

[4]As mentioned earlier, it is impossible to design commit protocols that are completely immune, in the non-blocking sense, to both site and link failures. Even the so called non-blocking protocols are prone to blocking under certain (perhaps highly unlikely) failure conditions.

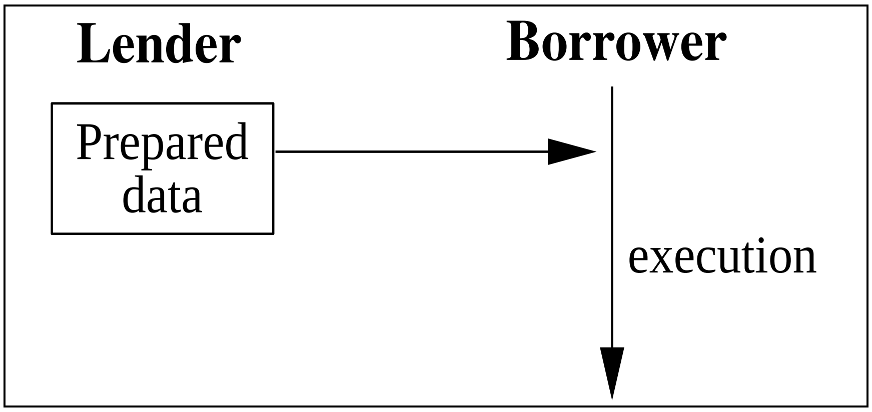

Figure 3.14: Optimistic Commit Protocol

3.6 Optimistic Commit Processing

In virtually all of the protocols described in the previous sections, a cohort that reaches the prepared state can release all of its read locks.However, it has to retain all its update locks until it receives the global decision from the master—this retention is fundamentally necessary to maintain atomicity. More importantly, the lock retention interval is *not bounded* since the time duration that a cohort is in the prepared state can be arbitrarily long (for example, due to network delays). If the retention period is large, it may have a significant negative effect on performance since other transactions that wish to access this (prepared) data are forced to block until the commit processing is over. It is important to note that this `data-blocking` is completely *orthogonal* to the `failure-blocking` (because of failures) as was discussed in Section 3.2.3. That is, in all commit protocols, including 3PC, transactions may become `data-blocked` waiting for (prepared) data to be unlocked. Moreover, such `data-blocking` occurs during normal processing whereas `failure-blocking` occurs only during failure situations.

To address the above issue of (prepared) `data-blocking`, a new version of the 2PC protocol was designed [28] , in which transactions requesting data items held by other transactions in the prepared state are allowed to access this data. That is, prepared cohorts *lend* uncommitted data to concurrently executing transactions (Figure 3.14). In this context, two situations may arise:

Lender Receives Decision First: Here, the lending cohort (i.e., the prepared cohort) receives its global decision before the borrowing cohort has completed its local execution. If the global decision is to commit, the lending cohort completes its processing in the normal fashion. If the global decision is to abort, then the lender is aborted in the normal fashion; in addition, the borrower is also aborted since it has utilized inconsistent data.

Borrower Completes Execution First: Here, the borrowing cohort completes its execution before the lending cohort has received its global decision. The borrower is now "put on the shelf", that is, it is made to wait and not allowed to send a WORKDONE message to its master. This means that the borrower is not allowed to initiate the processing that could eventually lead to its reaching the prepared state. Instead, it has to wait until the lender receives its global decision. If the lender commits, then the borrower is "taken off the shelf" and allowed to send its WORKDONE message. However, if the lender aborts, then the borrower is also aborted immediately since it has utilized inconsistent data.

In summary, the protocol allows transactions to access uncommitted data held by prepared transactions in the "optimistic" belief that this data will eventually be committed. We will hereafter refer to this protocol as **OPT**. Note that the basic structure of OPT is very similar to that of 2PC. Further, as the borrowing is a local (intra-site) phenomenon, the additional overheads of OPT are negligible.

3.6.1 Aborts in OPT do not Cascade

An important point to note here is that OPT's policy of using uncommitted data is generally *not* recommended in database systems since this can potentially lead to the well-known problem of *cascading aborts* if the transaction whose dirty data has been accessed is later aborted. However, for the OPT protocol, this problem is alleviated due to two reasons:

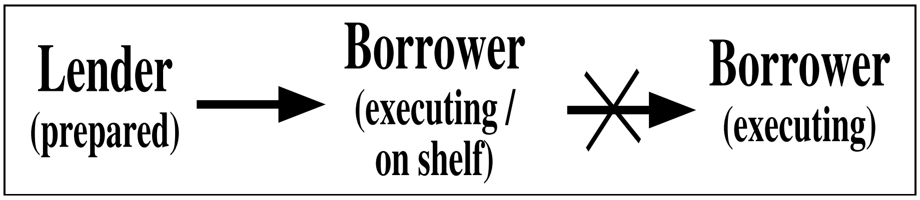

Figure 3.15: "Only a lender or a borrower be"

1. The lending transaction is typically expected to commit because (a) the lending cohort is in prepared state and cannot be aborted due to local data conflicts, and (b) the sibling cohorts are also expected to eventually vote to commit since they have survived all their data conflicts that occurred prior to the initiation of the commit protocol. In fact, if we assume that a lock based concurrency control mechanism such as, for example, 2PL, is used, it is easy to verify that there is *no* possibility of sibling cohorts aborting, during the commit processing period, due to serializability considerations. Therefore, an abort vote can arise only due to other reasons such as violation of integrity constraints, software errors, system failure, etc. We will hereafter use the term "surprise" aborts to refer to this type of aborts.

2. Even if the lending transaction does eventually abort, it only results in the abort of the immediate borrower and does not cascade beyond this point. This is because the borrower is not in the prepared state—the only situation in which uncommitted data can be lent. In other words, a borrower cannot simultaneously be a lender (Figure 3.15). In short, the abort chain is bounded and is of length one (of course, if an aborting lender has lent to multiple borrowers, then all of them will be aborted, but the length of each abort chain is limited to one).

3.6.2 Integrating Prior 2PC Optimization with OPT

Apart from the optimistic data access described above, the following features can also be included in the OPT protocol:

Presumed Abort/Commit : The PA and PC optimizations discussed earlier for 2PC (Section 3.3 and 3.4) can also be used in conjunction with OPT to reduce the protocol overheads. We consider both options in our experiments.

Non-blocking OPT : Our description of OPT above assumed a 2PC protocol as the basis. However, the OPT approach can be applied directly to the 3PC protocol as well. We evaluate the performance of this protocol also in our experiments.

Virtually all of the above optimizations can be integrated with an OPT implementation to produce *enhanced* performance. In particular, OPT is attractive with protocols such as *3PC* since they extend the period during which data is held in the prepared state.

3.6.3 System Integration

We now comment on the implementation issues related to the OPT protocol:

1. The lock manager at each site must be modified to permit borrowing of data held by prepared cohorts.

2. The lock manager must keep track of the cohorts that have borrowed prepared data so that, if the lender aborts, the borrowers can also be aborted.

Lender 1	Lender 2	...
Borrower 1	Borrower 2	...

Figure 3.16: Lender-Borrower List

One mechanism to accomplish this could be by maintaining a list where each element consists of a lender-borrower pair (Figure 3.16). Whenever a data item is lent to some executing cohort, the transaction ID pair of the lender and the borrower is entered into this list. (The lock held on the data item is updated as requested by the borrower). If the lender commits, the lender-borrower entry can be safely removed from the list. On the other hand, if the lender aborts the lender-borrower entry is removed from the list and in addition the borrower cohort is also aborted (which in turn will release the lock on the borrowed data item).

3. For a borrower cohort that finishes execution before its lenders have received their global decision, the local transaction manager must not send a *WORKDONE* message until the fate of its lenders is determined.

The above modifications do not appear difficult to incorporate in current database system software. Moreover, as shown later in our experiments, the performance benefits that can be derived from these changes suggest that it is worthwhile to make the effort of implementing them.

We believe that a systematic performance study of the various commit protocols would be of benefit to distributed database system designers—we address this issue in this work with respect to secure real time distributed databases. In addition, we also study the performance implications of supporting the non-blocking functionality. We also evaluate the new high-performance protocol (RT-OPT) [28] that is easy to implement and incorporate in current systems, and can be combined with a variety of other commit protocols including the PA, PC, and three-phase commit protocols.

CHAPTER

4

Real-Time Secure Commit Processing

The conventional transaction commit protocols cannot be directly used in the real-time environment, because they fail to take into consideration the real-time nature of the transactions. Therefore, the conventional commit protocols need to be modified to cater to the specific requirements of the real-time transactions. In this chapter, we discuss the issues involved in adapting the conventional commit protocols to the real-time environment. We also describe a real-time commit protocol, called RT-OPT, that is similar to OPT in its basic design but incorporates additional optimizations that cater to the special features of the real-time domain.**Most of the material in this chapter is taken from [30]**

4.1 Firm Deadline Distributed Transactions

As mentioned earlier, in a firm deadline real-time application framework, the transactions that miss their deadlines are considered to be worthless (and may even be harmful if executed to completion) and are immediately "killed" and discarded from the system without being executed to completion. Therefore, in an ideal distributed RTDB system, the master and all the cohorts of a transaction should agree to commit or abort before the transaction deadline expires. However, in practical distributed RTDB systems, it is impossible to provide such guarantees because of the possibility of communication link or site failures and arbitrary message delays. In addition, there are inherent complexities involved in achieving a global time-clock for the distributed system—synchronization of the clocks at various sites usually is a difficult task. Therefore, to avoid the inconsistencies in such cases, in a firm deadline RTDB system, a transaction is said to be **committed** if the master has reached the commit decision (that is, force-written the `COMMIT` log record to the disk) before the expiry of the deadline irrespective of whether the cohorts have received the decision by the deadline, that is, global time is viewed from the master's perspective. Note that once the master has reached the commit decision, this decision eventually will be conveyed to the cohorts even if the deadline has expired in the process. In other words, the master must reach the decision before the expiry of the deadline. If the master is unable to reach a commit decision before the deadline expires, it makes an abort decision, i.e., it immediately aborts the transaction and notifies the cohorts about the decision. Due to arbitrary message delays, a cohort may *not* receive the final decision before the deadline expires, even though the master had dispatched it before the expiry of the deadline. If a cohort eventually receives the final decision after the expiry of the deadline, all that happens is that access to data in prepared state is prevented even beyond the deadline until the decision is received by the cohort, and other transactions which would normally expect the data to be released by the deadline only experience a delay. Typically the master is responsible for returning the results of a transaction to the invoker of the transaction. From this discussion, it is clear that such results will have been generated before the deadline if a transaction commits.

4.2 Priority Inversion

The commit protocols described in Chapter 3 were designed for conventional database systems where transaction throughput or response time is the primary performance metric. In real-time database systems, however, satisfaction of transaction timing constraints is the primary goal. Therefore, the scheduling policies at the various resources (both physical and logical) in the system can be reasonably expected to be priority-driven with the priority assignment scheme being tuned to minimize the number of missed deadlines and to prevent covert channels. The conventional commit protocols, however, do not take transaction priorities into account. This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable.

Priority inversion is usually prevented by resolving all conflicts in favor of transactions with higher priority. At the CPU, for example, a scheduling policy such as Priority Preemptive Resume ensures the absence of priority inversion. For a non-preemptable resource such as the disk, although it is not possible to completely eliminate priority inversion due to physical device restrictions, at least it is possible to *bound* the inversion period by ensuring that the wait queue for the device is processed in priority order. Even at the network, it is easy to ensure that messages of high priority transactions are transmitted before those of other lower priority requesters. With respect to data access also, priority-based pre-emptive concurrency control algorithms such as 2PL-HP have been developed.

Removing priority inversion in the commit protocol, however, is *not* fully feasible. This is because, once a cohort reaches the prepared state, it has to retain all its data locks until it receives the global decision from the master—this retention is fundamentally necessary to maintain atomicity. Therefore, if a high priority transaction requests access to a data item that is locked by a prepared cohort of lower priority, it is not possible to forcibly obtain access by preempting the low priority cohort. In this sense, the commit phase in a distributed real-time database system is *inherently* susceptible to priority inversion. More importantly, the priority inversion is *not bounded* since the time duration that a cohort is in the prepared state can be arbitrarily long (for example, due to network delays). If the inversion period is large, it may have a significant effect on performance.

4.3 Real-Time Secure Commit Protocols

The conventional commit protocols suffer from two problems: (a) they are not deadline cognizant and hence are not capable of meeting the transaction deadline semantics as described in Section 4.1, and (b) they do not make use of the transaction priorities at the physical and logical resources, rendering them inefficient in the real-time domain. Even though the priority inversion arising due to prepared cohorts cannot be eliminated in these commit protocols, in order to be useful in the real-time environment, these commit protocols are modified to incorporate the following:

- In order to avoid covert channels the priority assigned to a transaction is a vector consisting of the level and the deadline, thereby giving priority to the lower level first and then the deadline.

- The commit messages have the priorities of the transactions for which they are generated, so that the messages of a higher priority transaction can be given preference over the lower priority transactions at the CPU. In addition, processing of the data pages of a higher priority transaction at the CPU is given preference over the messages of lower priority transactions.

- The master ensures that a decision about the transaction is reached by the deadline. If the deadline expires before the master had dispatched the PREPARE messages to the cohorts, it *unilaterally* decides to abort the transaction and sends the ABORT messages to the cohorts. On the other hand, if the deadline expires after dispatching the PREPARE messages, but before the master had reached a decision (i.e., all votes have not yet been received), the master makes an abort decision and the abort processing of the transaction is done in accordance with the protocol being used. For example, if PA is being used, the master writes the `ABORT` log record, sends the ABORT messages to all the cohorts, and forgets the transaction without waiting for the ACKs from the cohorts.

These modifications to the conventional commit protocols ensure that the deadline semantics of a transaction as described in Section 4.1 are met. Also, the real-time variant of the commit protocols exploit the priorities of the transactions while contending for the resources for message processing, etc. Hereafter we will not use the term "real-time variant" explicitly while referring to the real-time variants of the commit protocols. If the non-real-time (conventional) versions of the commit protocols are to be meant, we will explicitly state so.

4.4 Real-Time Optimistic Commit Protocol

As discussed in Section 4.2, removing priority inversion in a commit protocol is *not* fully feasible because the lower priority cohorts that are in the prepared state cannot be preempted by the higher priority cohorts. This may have a significant effect on the system performance. To address this issue, a modified version of the 2PC protocol was designed [28], wherein transactions requesting data items held by lower priority transactions in the prepared state are allowed to access this data. That is, lower priority prepared cohorts *lend* uncommitted data to the higher priority cohorts. In this context, two situations may arise:

Lender Receives Decision First: Here, the lending cohort (i.e., the prepared cohort) receives its global decision before the borrowing cohort has completed its local execution. If the global decision is to commit, the lending cohort completes its processing in the normal fashion. If the global decision is to abort, then the lender is aborted in the normal fashion; in addition, the borrower is also aborted since it has utilized inconsistent data.

Borrower Completes Execution First: Here, the borrowing cohort completes its execution before the lending cohort has received its global decision. The borrower is now "put on the shelf", that is, it is made to wait and not allowed to send a WORKDONE message to its master. This means that the borrower is not allowed to initiate the processing that could eventually lead to its reaching the prepared state. Instead, it has to wait until either the lender receives its global decision or its own deadline expires, whichever occurs earlier. In the former case, if the lender commits, then the borrower is "taken off the shelf" and allowed to send its WORKDONE message. However, if the lender aborts, then the borrower is also aborted immediately since it has utilized inconsistent data. In the latter case (deadline of a borrower expires while waiting on the shelf), the borrower is killed in the normal manner.

In summary, the protocol allows transactions to access uncommitted data held by lower priority prepared transactions in the "optimistic" belief that this data will eventually be committed. Hereafter we will refer to this protocol as **RT-OPT**.

It was mentioned in Section 4.1 that if a cohort does not receive the decision from its master by the deadline, it will continue to hold the data in the prepared state beyond its deadline, thus preventing access to the data by other transactions. Therefore, a transaction that would normally expect the data to be released by the deadline will experience a delay. Using RT-OPT protocol solves this problem to a considerable extent as the data in the prepared state is made available to other transactions. If the cohort in the prepared state finally receives a commit decision (after the deadline has expired), only the transactions that were waiting on the shelf for the lender cohort to commit when they were aborted—perhaps because their deadlines expired—are affected (mostly these transactions otherwise also would have been killed, as their deadlines were close enough). Other borrower transactions are ignorant about the fact that the lender cohort actually received the decision only after its deadline because these transactions never went on the shelf. On the other hand, if the cohort in the prepared state finally receives an abort decision, all that happens is that the restarts of the transactions that had accessed uncommitted data is delayed, which otherwise would have occurred at the deadline expiry of the prepared cohort.

The primary motivation, as described earlier, for permitting access to uncommitted data was to reduce priority inversion. However, if we believe that lender transactions will typically commit, then this idea can be carried further to allowing *all* transactions, including low priority transactions, to borrow uncommitted data. This may further help in improving the real-time performance since the waiting period of transactions is reduced, and is therefore incorporated in RT-OPT. Also by permitting access to uncommitted data we can avoid the covert channel possibility present in all the other protocols, since there the locks will not be released by the higher level (i.e. lower priority) when they have entered the prepared mode.

4.4.1 AdditionalFeatures of RT-OPT

The following features have also been included in the RT-OPT protocol since we expect them to improve its real-time performance:

Active Abort : In the basic 2PC protocol, cohorts are *passive* in that they inform the master of their status only upon explicit request by the master, that is, if the cohort is aborted after it had dispatched the WORKDONE message, it will inform the master only on receiving the PREPARE message from the master. In a conventional distributed DBMS, after a cohort has finished its work and sent the WORKDONE message to the master, it cannot be aborted due to local data conflicts or serializability violations (assuming a lock-based concurrency control mechanism). However, in a distributed RTDB system, even such a cohort can be aborted due to higher priority conflicting transactions (provided the cohort is not already in the prepared state). Therefore, in a real-time situation, it may be better for an aborting cohort to immediately inform the master so that the abort at the other sites can be done earlier. Hence, cohorts in RT-OPT inform the master as soon as they decide to abort locally.

Silent Kill : For a transaction "kill", that is, an abort that occurs due to missing the deadline (before the master dispatched PREPARE messages), there is no need for the master to invoke the abort protocol since the cohorts of the transaction can independently realize the missing of the deadline (assuming global clock synchronization). Therefore, in RT-OPT, aborts due to deadline misses—before a cohort enters the prepared state—are done "silently" without requiring any communication between the master and the cohorts.

In addition, the PA and PC optimizations discussed earlier can be used in conjunction with RT-OPT . Also, RT-OPT can be used with 3PC protocol to provide both the non-blocking feature of 3PC and the high performance of RT-OPT. We consider all these combinations in our experiments.

4.4.2 No Cascading Aborts in RT-OPT

Even though access to uncommitted data is allowed, aborts in RT-OPT do not cascade (as was the case with OPT in the context of OLTP systems, described in Chapter 3). The problem of cascading aborts in RT-OPT is alleviated due to the following reasons:

1. The lending cohort is in prepared state and cannot be aborted due to local data conflicts (we assume a lock-based concurrency control mechanism). Therefore, if the lending transaction finally decides to abort, it does so because of two reasons (in addition to consistency violations): (a) the master has already sent the PREPARE messages, but the deadline expires before it is able to reach a decision (Section 4.1), or (b) any of the siblings of the cohort in the prepared state votes NO. Note that when a cohort is aborted, it immediately sends an ABORT message to master (Active Abort policy), and the master, after receiving an ABORT message from a cohort never sends the PREPARE messages to other cohorts. Therefore, the situations in which a sibling of the prepared cohort could vote NO can be the following: (i) the ABORT message sent by the sibling cohort was in *transit* when master dispatched the PREPARE messages, OR (ii) the PREPARE message sent by the master to the sibling cohort was in transit when the cohort was aborted. As the time during which a message is in transit is usually small compared to the transaction execution times, the above situations are unlikely to occur frequently. Hence, a lending transaction is typically expected to commit.

2. Even if the lender eventually does abort, it only results in the abort of the immediate borrower(s) and does not cascade beyond this point (since borrowers are not in the prepared state which is the only situation in which uncommitted data can be accessed).

In short, the abort chain for RT-OPT is bounded and is of length one (of course, if an aborting lender has lent to multiple borrowers, then all of them will be aborted, but the abort chain for each borrower is bounded as described above).

We have in our research focused on quantitative analysis of the real-time performance of the *standard* mechanisms for maintaining distributed transaction atomicity *without* relaxing the transaction correctness criterion and taking care of *security*.

CHAPTER

5

SRTDB Simulation Model

To evaluate the performance of the various real-time commit protocols described in the previous chapter, we developed a detailed simulation model based on an *open* queueing model of a distributed real-time database system. The simulation model is based on the distributed database model described in [28] The model consists of a database that is distributed, in a non-replicated manner, over N sites connected by a network. Each site in the model has six components: a *source* which generates transactions and has an access manager which will take care of assigning data only of levels permitted to be accessed by the respective level transactions; a *transaction manager* which models the execution behavior of the transaction; a *concurrency control manager* which implements the concurrency control algorithm; a *resource manager* which models the physical resources; a *recovery manager* which implements the details of commit protocols; and a *sink* which collects statistics on the completed transactions. In addition, a *network manager* models the behavior of the communications network. Figure 5.1 shows a slightly more detailed view of these components.

In the following sections, we describe various components of the simulation model. Subsequently, we describe the execution pattern of a typical transaction. The various parameters used in the model are given in Table 5.1.

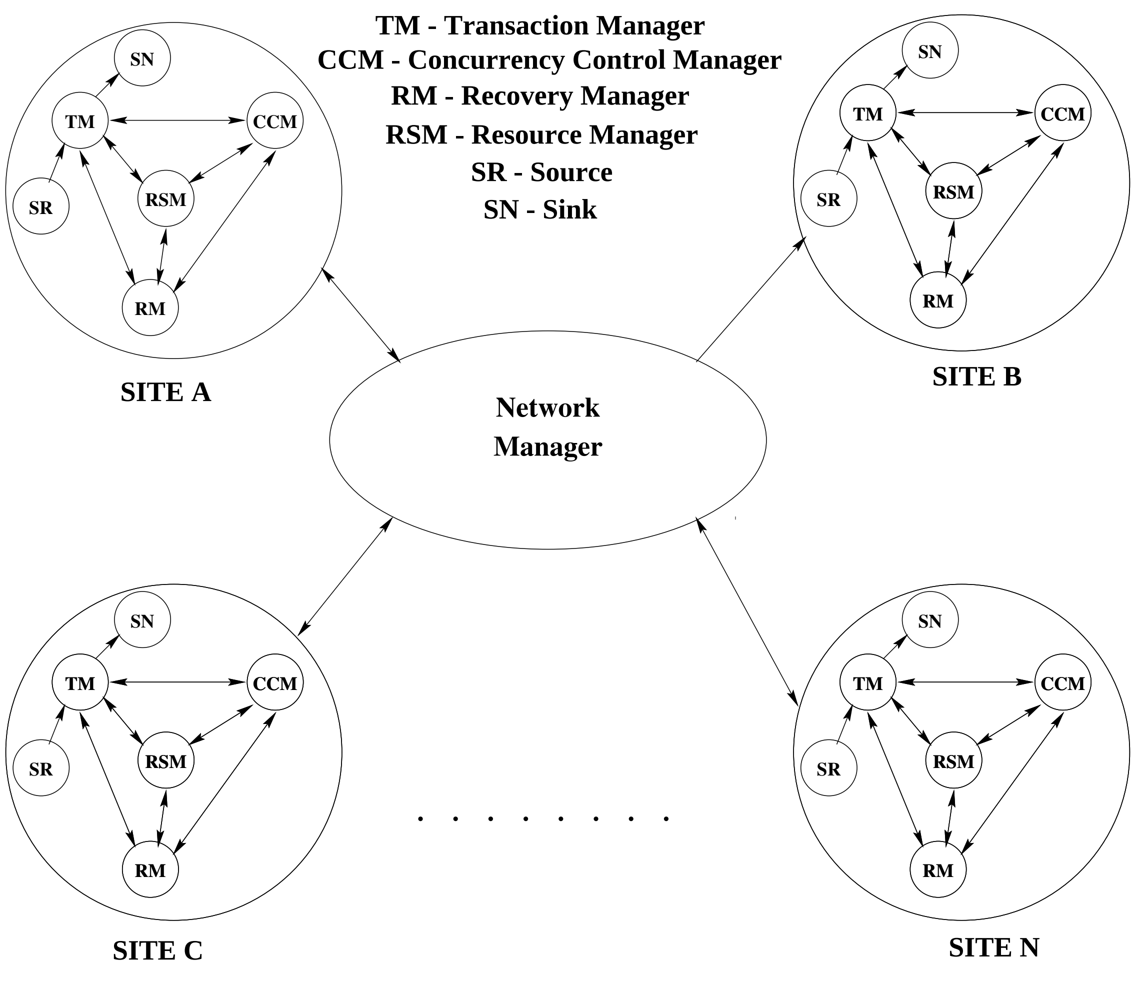

Figure 5.1: Distributed DBMS Model Structure

Table 5.1: Simulation Model Parameters

$NumSites$	Number of sites in the database
$numFiles$	Number of files in the database
$numPages$	Number of pages in the database
$ClassLevels$	Number of Classification Levels
$ArrivalRate$	Transaction arrival rate / site
$ClearLevels$	Number of Clearance Levels
$SlackFactor$	Slack Factor in Deadline formula
$execPattern$	Transaction Type (Sequential or Parallel)
$DistDegree$	Degree of Distribution
$CohortSize$	Average cohort size (in pages)
$UpdateProb$	Page update probability
$NumCPUs$	Number of processors per site
$NumDataDisks$	Number of data disks per site
$NumLogDisks$	Number of log disks per site
$PageCPU$	CPU page processing time
$PageDisk$	Disk page access time
$MsgCPU$	Message send / receive time
$BufHit$	Probability of finding a requested page in the buffer

5.1 Database Model

The database is modeled as a collection of $numFiles$ files each of $numPages$ pages that have been assigned $ClassLevels$ and are uniformly distributed in a non-replicated fashion across all the $NumSites$ sites. There are transactions arriving at each of the sites in a continuous fashion in an independent Poisson stream at the rate of $ArrivalRate$, and each transaction is assigned a firm deadline using the formula $D_T = A_T + SlackFactor * E_T$, where D_T, A_T and E_T are the deadline, arrival time and execution time, respectively, of transaction T, while $SlackFactor$ is a constant that provides control over the tightness/slackness of transaction deadlines. The execution time is the total service time at the resources that the transaction requires for its execution. [1] Note that although the workload generator knows E_T, the system does *not* have any knowledge about E_T.

Each transaction in the workload has the "single master, multiple cohort" structure

[1]Since the resource time is a function of the number of messages and the number of forced-writes, which differ from one commit protocol to another, we compute the execution time assuming execution in a *centralized* system.

described in Section 3.1.1. The number of sites at which each transaction executes is specified by the $DistDegree$ parameter. The master and one cohort reside at the site where the transaction is submitted whereas the remaining $DistDegree - 1$ cohorts are set up at different sites chosen at random from the remaining $NumSites - 1$ sites. At each of the execution sites, the number of pages accessed by the transaction's cohort varies uniformly between 0.5 and 1.5 times $CohortSize$. These pages are chosen randomly from among the database pages located at that site. A page that is read is updated with probability $UpdateProb$. A transaction that is aborted due to a data conflict is restarted immediately and makes the same data accesses as its original incarnation.

5.2 System Model

The physical resources at each site consist of $NumCPUs$ processors, $NumDataDisks$ data disks, and $NumLogDisks$ log disks. There is a single common queue for the CPUs and the service discipline is Pre-emptive Resume, with preemptions being based on transaction priorities. Each of the disks has its own queue and is scheduled according to a Head-Of-Line (HOL) policy, with the request queue being ordered by transaction priority. The $PageCPU$ and $PageDisk$ parameters capture the CPU and disk processing times per data page, respectively. A data page request can be satisfied by the buffer manager, or the page may have to be accessed from the disk. $BufHit$ parameter gives the probability of finding a page in the buffer pool.

The communication network is simply modeled as a switch that routes messages since we assume a local area network that has high bandwidth. However, the CPU overheads of message transfer are taken into account at both the sending and the receiving sites. This means that there are two classes of CPU requests—local data processing requests and message processing requests. We do not make any distinction, however, between these different types of requests and only ensure that all requests are served in priority order. The CPU overheads for message transfers are captured by the $MsgCPU$ parameter.

5.3 Priority Assignment

As mentioned earlier, transactions in a RTDB system are typically assigned priorities in order to avoid covert channels and also to minimize the number of missed deadlines. In our model, all the cohorts of a transaction inherit the parent transaction's priority. Further, this priority, which is assigned at arrival time, is maintained throughout the course of the transaction's existence in the system (including the commit processing stage, if any). The messages sent by transactions also inherit the priority of the transactions. The transactions are assigned priority such that lower levels have higher prioirty than higher levels and within levels priority is based on Earliet Deadline wherein the transactions with earliar deadlines are considered to have higher priority over those with later deadlines.

5.4 Concurrency Control

The 2PL High Priority scheme is used for concurrency control of conflicting transactions. A concurrency control request from a high priority cohort for a data item held by a low priority cohort is satisfied by aborting the low priority cohort. If a low priority cohort requests a data item held by a high priority cohort, it is made to wait until the high priority cohort releases its lock from the data item (or allows optimistic access, as in RT-OPT). If a data item is locked by some cohort in the read mode, a cohort's request for sharing the lock is granted only if no cohort with a priority higher than that of the requesting cohort is waiting in the lock queue to access the data in an exclusive (write) mode. A cohort in the prepared state cannot be aborted due to concurrency reasons. On receipt of the PREPARE message from the master, a cohort releases all its read locks but retains its update locks until it receives and implements the global decision from the master. Due to the fact that 2PL-HP protocol is used to resolve the conflict between transactions, and that the transactions are assigned unique priorities at the arrival time, there is no possibility of the transactions forming a deadlock.

5.5 Execution Model

A transaction, at its arrival time, is assigned the set of sites where it has to execute , the files, data pages that it has to access at each of these sites and it is also assigned the clearance level. The deadline of the transaction is assigned based on the formula described in Section 5.1. The parameter $execPattern$ specifies whether the transaction will execute in a sequential or parallel fashion. The master is then started up at the originating site, forks off a local cohort and sends messages to initiate each of its cohorts at the remote participating sites. Each cohort makes a series of read and update accesses. A read access involves a concurrency control request to obtain access permission, followed by a disk I/O to read the page, followed by a period of CPU usage for processing the page. We do not explicitly model concurrency control cost parameters in our model since we assume that the overheads of performing concurrency control are small compared to data processing times. Update requests are handled similarly except for their disk I/O—the writing of the data pages takes place asynchronously after the transaction has committed. We assume sufficient buffer space to allow the retention of data updates until commit time.

The commit protocol is initiated when the transaction has completed its data processing. If the transaction's deadline expires either before this point, or before the master has written the global decision log record, the transaction is killed (the precise semantics of firm deadlines in a distributed environment are defined in Section 4.1). If the master has written the commit decision log record before the expiry of the deadline, the transaction is said to be committed.

5.6 Logging

With regard to logging costs, we explicitly model only *forced* log writes since they are done synchronously and suspend transaction operation until their completion. The cost of each forced log write is the same as the cost of writing a data page to the disk. The overheads of flushing the transaction execution log records (i.e., WAL) are not modeled since we do not expect them to affect the relative performance behavior of the commit protocols discussed in our study.

CHAPTER

6

SRTDB Experiments and Results

In order to evaluate the performance of the various commit protocols desribed in the Chapter 4 a number of experiments were conducted using the simulation model desribed in Chapter 5 using C++SIM on Sun UltraSparc machines running Solaris 3.1 operating system. **Some parts of the explanations of graphs in this chapter have been taken from [30]**

6.1 Performance Metric

The performance metric used here is the $MissPercent$ i.e. the number of transactions that are unable to complete execution within thier deadline . Besides we also plot the $Transactionthrougput$ since it will help observe when thrashing occurs , so that we can take some measures to ensure that the arrivals are limited to that particular Arrival rate above which thrashing sets in. And since these experiments are being performed to observe the effect of integrating security into the real time domain, we plot graphs of the fairness factor against the arrival rate. The fairness of each level 'i' is calculated as Fairness[i] = numCommitedTrans[i]/numInputTrans[i] where numCommitedTrans[i] refers to the number of transactions commited at level i and numInputTrans[i] refers to the total number of input transactions at level i. And in the plots fairness factor of level i is calulated as Fairness[i]/TotalFairness where TotalFairness is the sum of the fairness of all the levels. Also plots of levelwise $MissPercent$ and $RestartRatios$ are plotted. These

Table 6.1: Baseline Parameter Values

NumSites	8	*NumCPUs*	2
NumClassLevels	2	*NumClearLevels*	2
numFiles	8 files	*numPages*	300 pages
NumDataDisks	3	*NumLogDisks*	1
SlackFactor	4.0	*PageCPU*	5 ms
TransType	Sequential	*PageDisk*	20 ms
DistDegree	3	*MsgCPU*	5 ms
CohortSize	6 pages	*BufHit*	0.1
UpdateProb	1.0	*ArrivalRate*	0.5 - 20

plots will help us observe the effects of integrating security on the performance of the various levels of transactions.

Each of the experiments is initially run for 4000 transactions to help the system warm up, and then the run is until 20000 transactions are processed by the system.

The simulator was used to generate other statistical information such as resource utilisation, number of forced writes , page conflict probability etc. which help explain the *MissPercent* graphs behaviour.

6.2 Experiment 1: Baseline Experiment

We began the performance evaluation of the secure real-time commit protocols by first developing a baseline experiment. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. The values of the various parameters for the baseline experiment are listed in Table 6.1. These values were chosen to ensure significant levels of both resource contention (RC) and data contention (DC) in the system, thus helping to bring out the performance differences between the various commit protocols, without having to generate very high transaction arrival rates.

For the baseline experiment, Figures 6.1a and 6.1b show the MissPercent behavior under normal load and heavy load conditions, respectively.

We see from Figure 6.1c that the peak throughput is achieved at an arrival rate of approximately 3 transactions per second for 2PC and 3PC. These values of arrival rates correspond to transaction MissPercent range of 30 percent to 40 percent, for all these protocols. Thus, we observe that the systems is being utilized to the maximum as long as the MissPercent value is in the range of 30 percent to 40 percent, beyond which it starts thrashing.

Moving on to the relative performance of 2PC and 3PC, we observe from Figures 6.1a and 6.1b that there is a noticeable but not large difference between their performance at normal loads. The difference arises from the additional message and logging overheads involved in 3PC. Under heavy loads, however, the performance of 2PC and 3PC is virtually identical. This is explained as follows: Although their commit processing is different, the *abort* processing of 3PC is identical to that of 2PC. Therefore, under heavy loads, when a large fraction of the transactions wind up being killed (aborted) the performance of both protocols is essentially the same. Since their performance difference is not really large for normal loads also, it means that, in the real-time domain, the price paid during normal processing to purchase the nonblocking functionality is comparatively modest.

Shifting our focus to the PA and PC variants of the 2PC protocol, we find that their performance is only marginally different to that of 2PC. Thus, we see that these optimizations fail to provide tangible performance benefits. We would have, however, expected PC to perform at least slightly better than 2PC at normal loads, because the utilization of the resources is reasonably high at normal loads and most of the transactions are finally committed—the conditions favorable to PC. Similarly, we would have expected PA to perform better at high loads, because most of the transactions are aborted. The reason that PA and PC do not show much improvement in the performance, is that the performance in an SRTDB system is measured in *boolean* terms of meeting or missing the deadline. So, although PA and PC do reduce overheads under abort and commit conditions, respectively, all that happens is that the resources released by this reduction only allow executing transactions to execute further before being restarted or killed but is not sufficient to result in many more *completions*. This was confirmed by measuring the number of force-writes and the number of acknowledgements, on a per transaction basis.

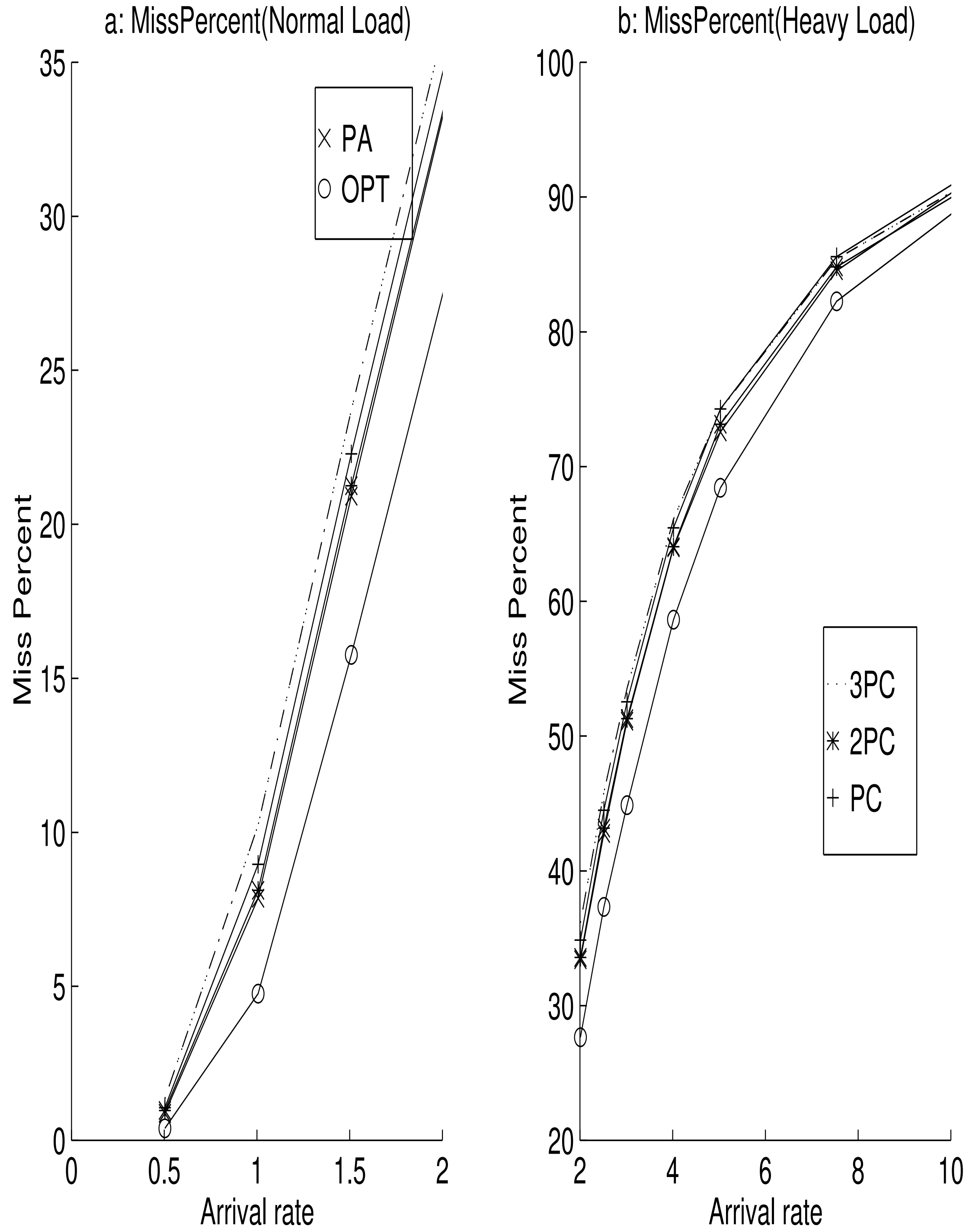

Figure 6.1: (a,b) Baseline Experiment (Sequential, RC+DC)

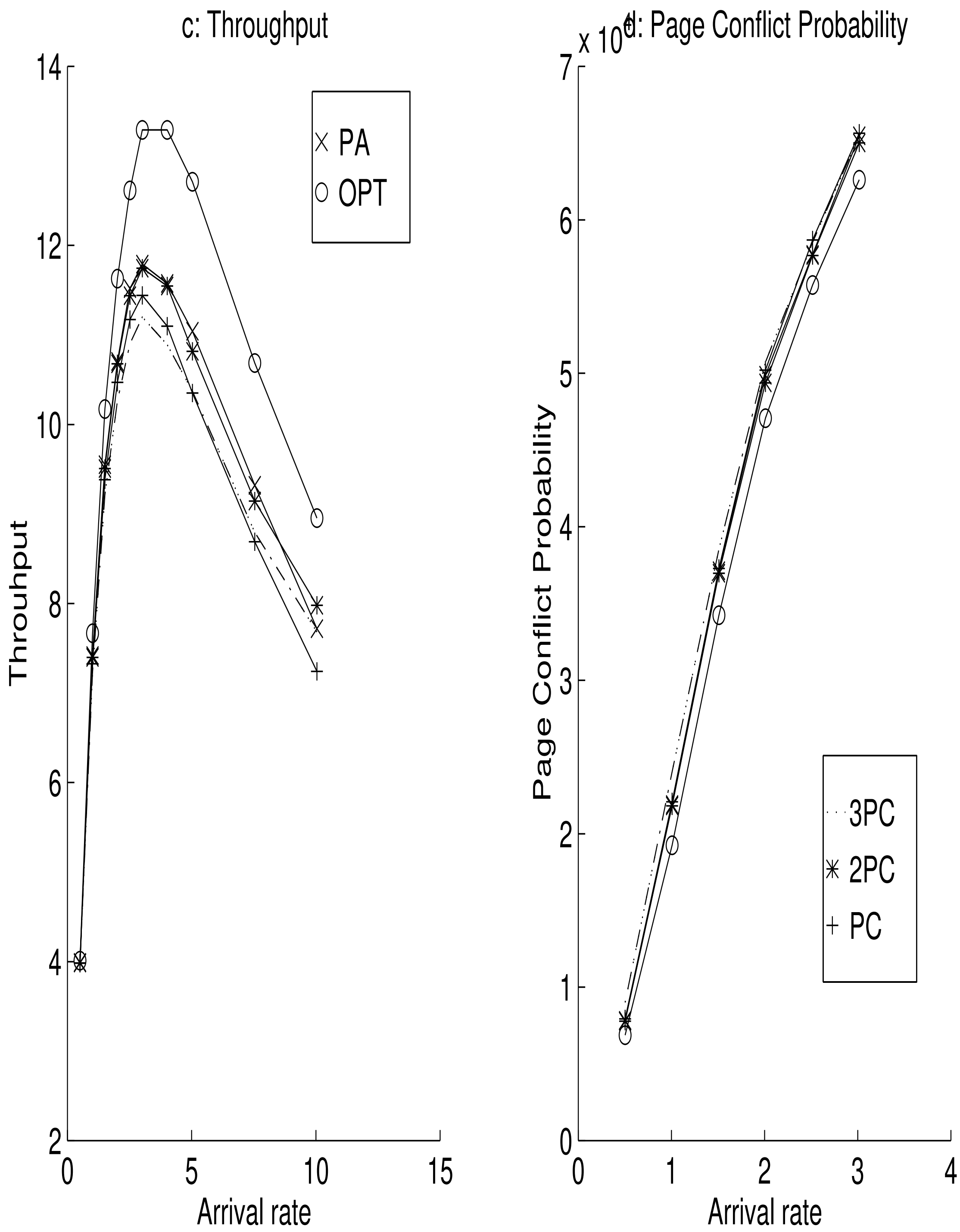

Figure 6.1: (c,d) Baseline Experiment (Sequential, RC+DC)

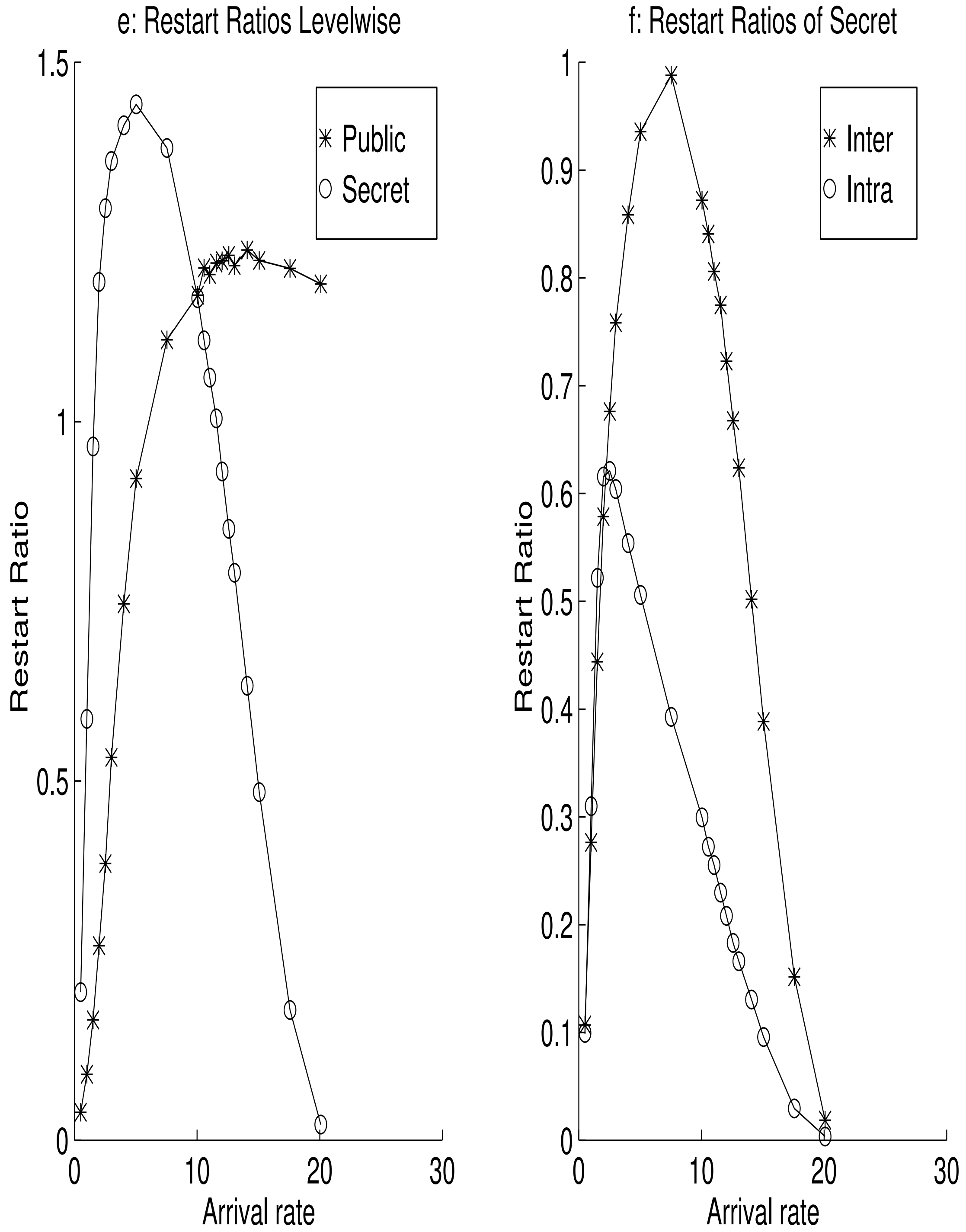

Figure 6.1: (e,f) Baseline Experiment (Sequential, RC+DC)

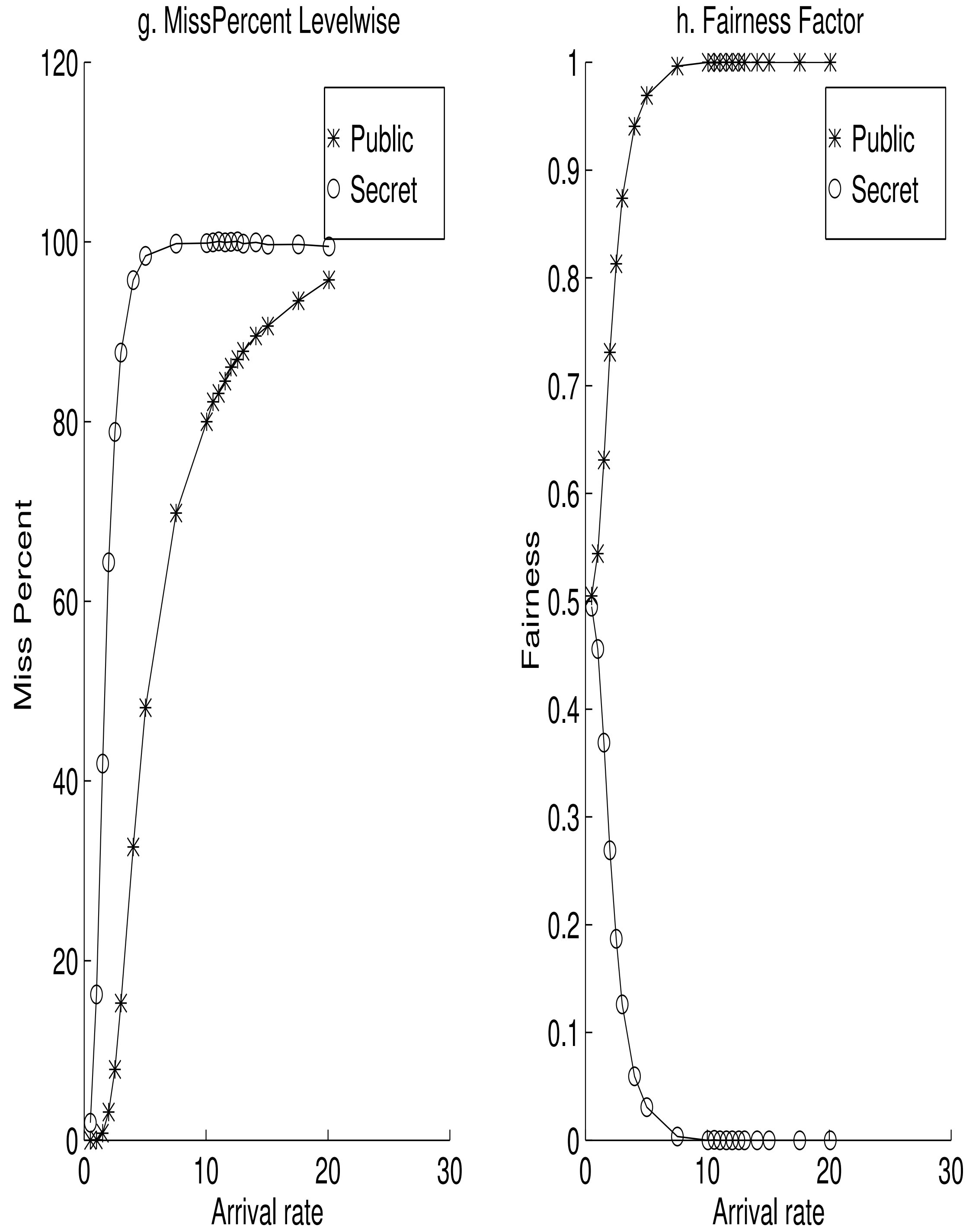

Figure 6.1: (g,h) Baseline Experiment (Sequential, RC+DC)

Moreover, while PA always does slightly better than 2PC, PC actually does slightly worse than 2PC since PC has higher overheads than 2PC for aborts.

Finally, turning our attention to the new protocol, RT-OPT, we observe that its performance is considerably better than that of the standard algorithms over most of the loading range and especially so at normal loads. An analysis of its improvement showed that it arises primarily from the optimistic access of uncommitted data and from the active abort policy. The silent kill optimization (not sending abort messages for aborts arising out of deadline misses), however, gives only a very minor improvement in performance. At low loads, this is because the number of deadline misses are few and the optimization does not come into play; at high loads, the optimization's effect is like that of PA and PC for the standard 2PC protocol—although there is a significant reduction in the number of messages, the resources released by this reduction only allow transactions to proceed further before being restarted but does not result in many more *completions*.

RT-OPT helps improve the performance of the system due to the following reason: By allowing access to the uncommitted data of the prepared cohorts, it reduces the data contention in the system. This is clear from Figure 6.1d where the page conflict probability (the probability that a page being requested by a cohort is already held in a conflicting mode by another cohort) for RT-OPT is less than that for all other commit protocols. The Fairness performance results for 2PC is shown in the Figure 6.1h where it is observed that the *Secret* level is suffering more due to the priority given to the lower level Public Transactions (in order to avoid Covert Channels). It is abvoius therefore that some measure should be taken so that fairness is ensured for the Secret Transactions by say restricting the entry of Public Transactions after a certain amount of time as suggested in [31].

This is also obvious from the Figure 6.1g and Figure 6.1e which show the levelwise $MissPercent$ and levelwise *RestartRatios* respectively. As far as *RestartRatios* are concerned, it is observed from the Figure 6.1e that after a certain arrival rate the curve begins to drop, which can be explained as follows , since there is resource contention as well we see that transactions are getting heldup while waiting in the queues for resources and they eventually would get killed due to the expiry of their deadlines.

The Figure 6.1f shows the *RestartRatios* within the Secret level due to *InterLevel* and *Intralevel* conflicts, wher *InterLevel* conflicts are conflicts between levels and *IntraLevel* conflicts are conflicts within the same level due to deadlines. From this graph it can be seen that the *InterLevel* conflicts are higher than *IntraLevel* conflicts due to the Public level being given a higher priority to avoid covert channels.

6.3 Experiment 2: Pure Data Contention

The goal of our next experiment was to isolate the influence of *data contention* (DC) on the real-time performance of the commit protocols. Therefore, for this experiment, the physical resources (CPUs and disks) were made “infinite”, that is, there is no queueing for the these resources. The other parameter values are the same as those used in Experiment 1. The MissPercent , throughput and Page Conflict probability performance results for this system configuration are presented in Figures 6.2a through 6.2d, and the restart ratios and Fairness results are shown in Figures 6.2e through 6.2h.

We observe in these figures that the relative performance of the various protocols remains qualitatively similar to that seen under resource-cum-data contention (RC+DC) in the previous experiment. Under pure DC, however, the performance is limited only by data contention. Therefore, although the MissPercent under pure DC is lower than that under RC+DC, the effect of the commit phase on MissPercent performance is felt to a greater extent as the data held in the commit phase is not accessible to other transactions, thus increasing the data contention. We also observe the performance of RT-OPT in this experiment is much superior to standard commit protocols over the entire loading range as compared to the previous experiment. This is for the same reason explained above. RT-OPT allows access to the data held by the transactions in the commit phase, thus reducing the data contention in the system.

An important observation here is that while resource contention can be reduced by purchasing more and/or faster resources, there exists no equally simple mechanism to reduce data contention. It should be noted here that while abundant resources may not

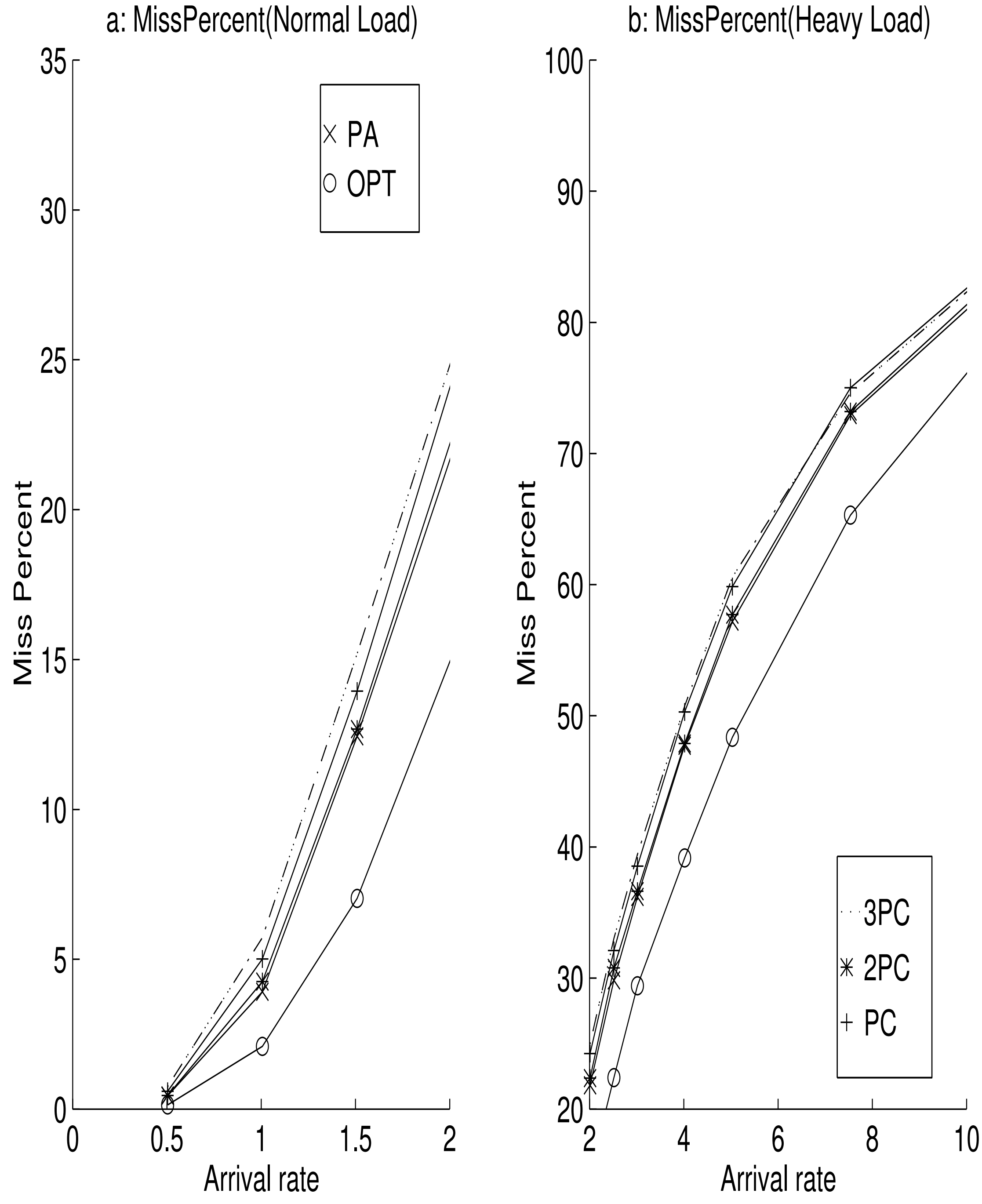

Figure 6.2: (a,b) Baseline Experiment (Sequential, PureDC)

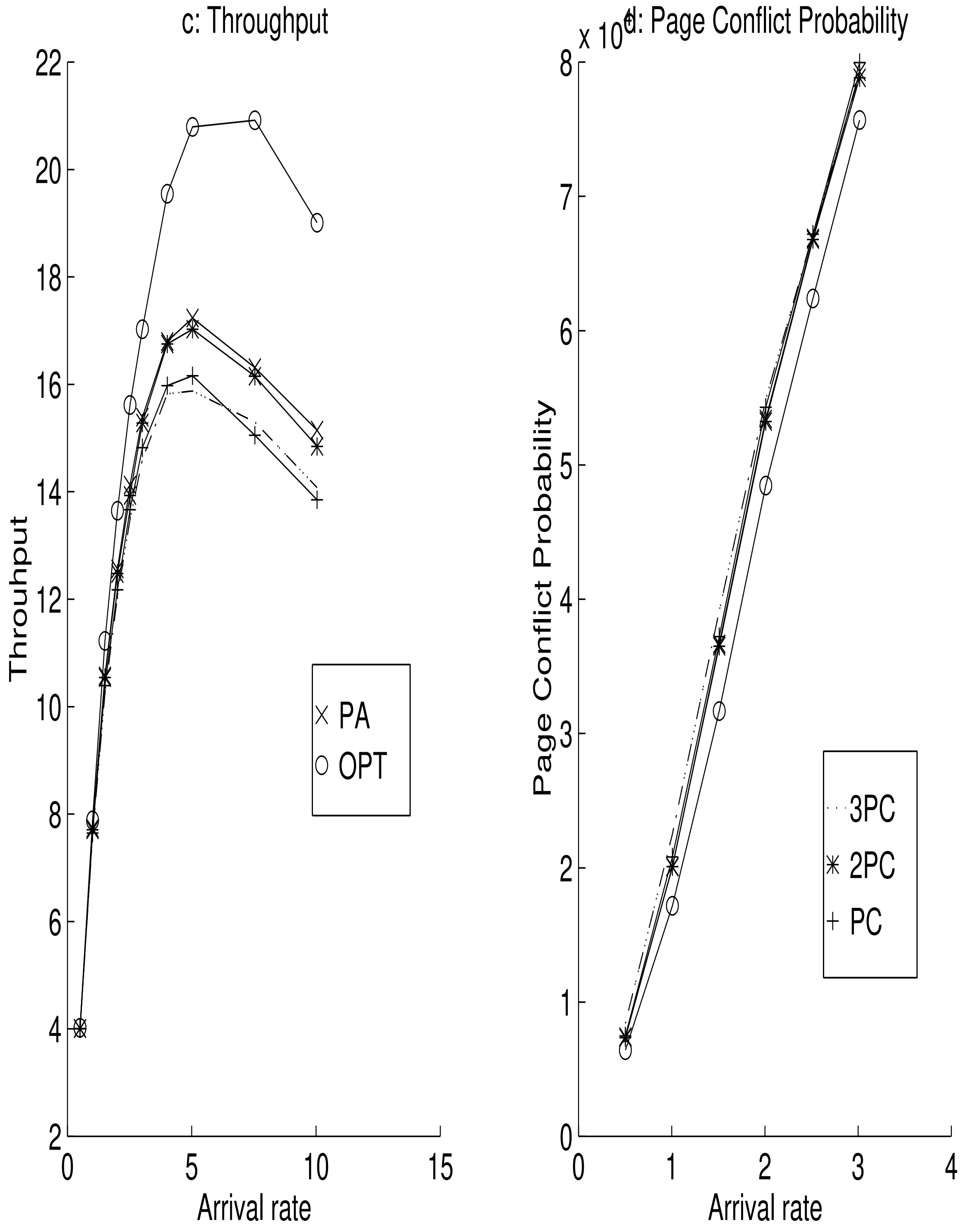

Figure 6.2: (c,d) Baseline Experiment (Sequential, PureDC)

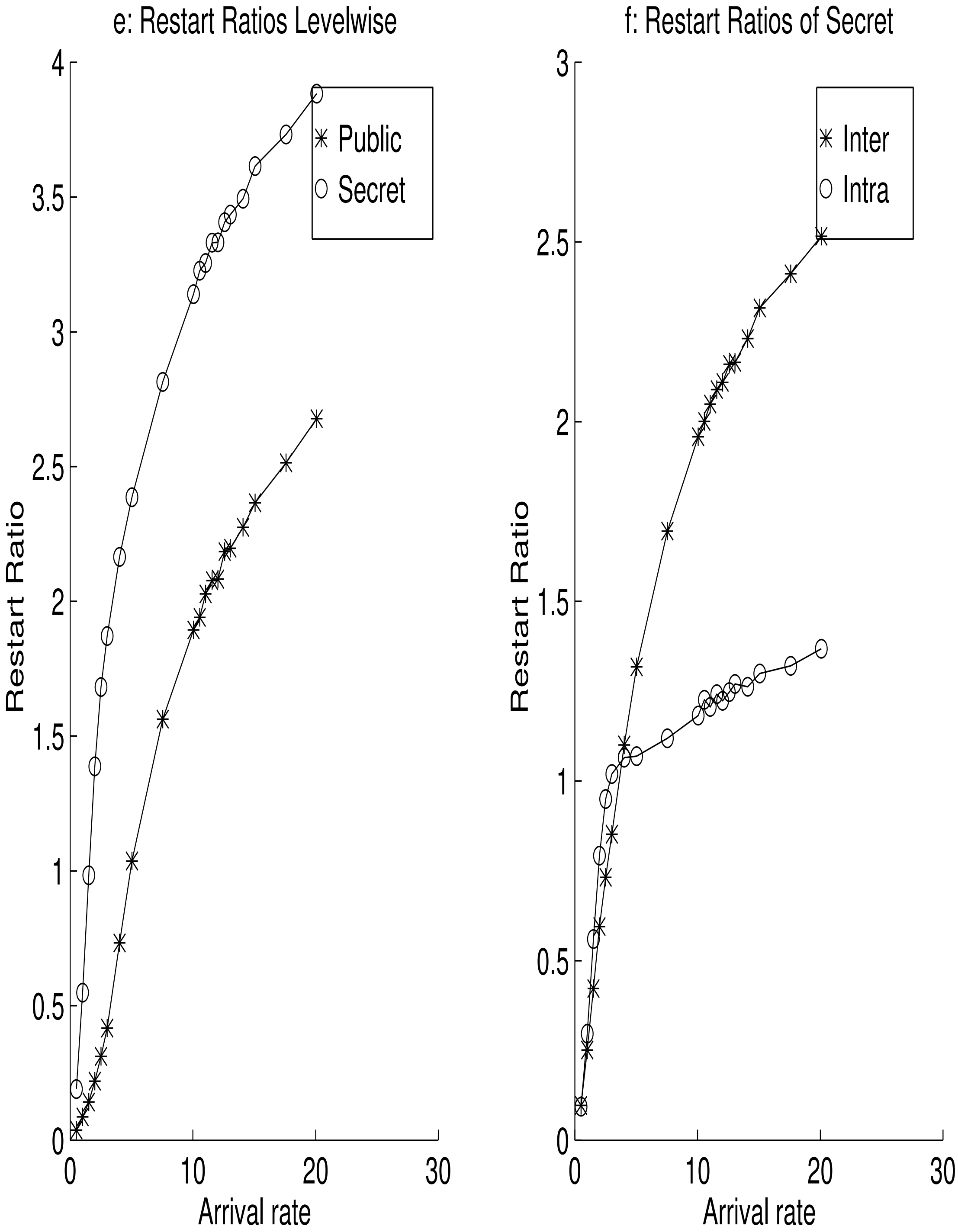

Figure 6.2: (e,f) Baseline Experiment (Sequential, PureDC)

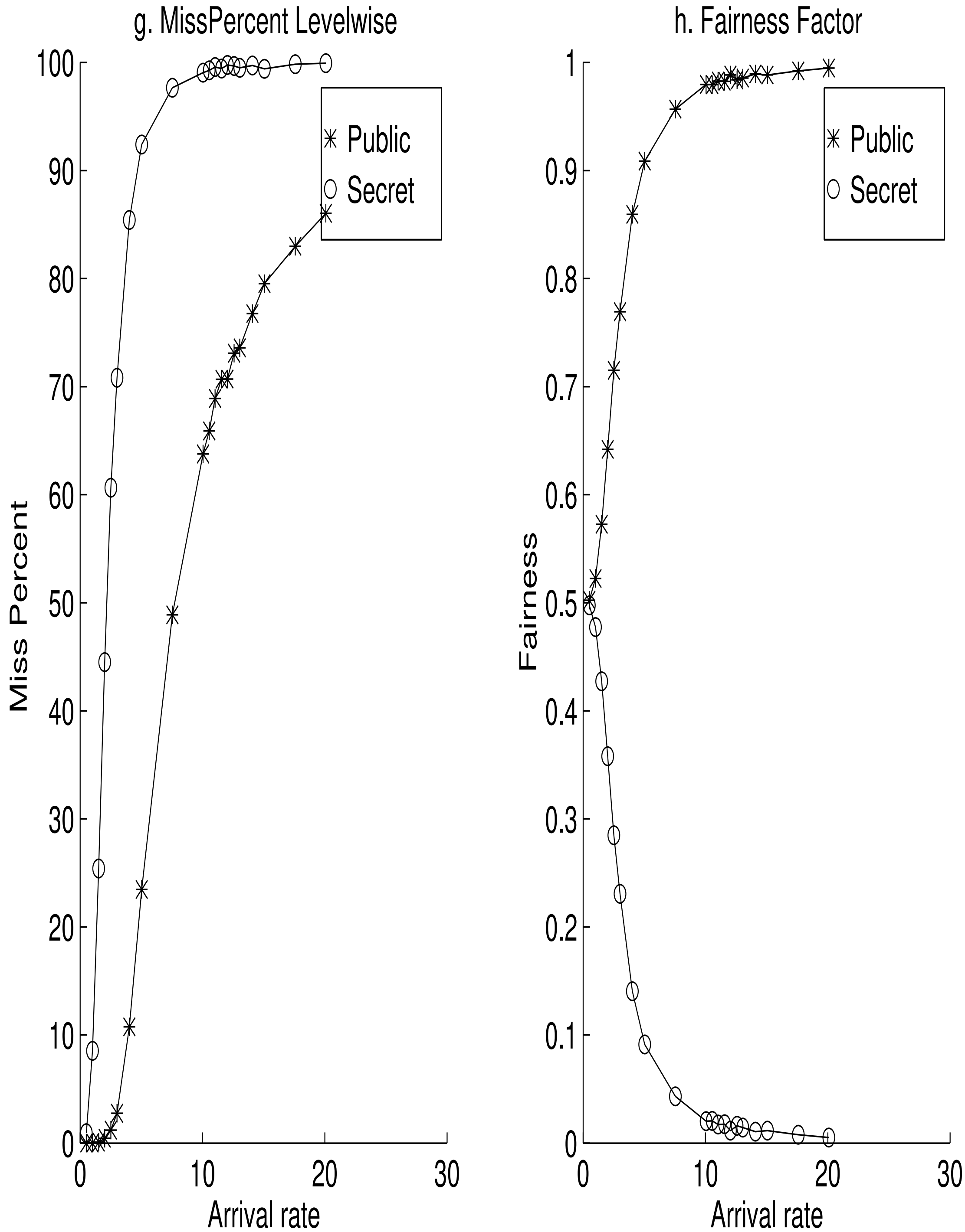

Figure 6.2: (g,h) Baseline Experiment (Sequential, PureDC)

be expected in conventional database systems, that may not be the case with RTDB systems since many real-time systems are designed to handle transient heavy loading.

Here since the resources are unlimited it can be observed from the Figure 6.2e that the *RestartRatios* are continuously increasing with the increased arrival rates.

6.4 Experiment 3: Pure Resource Contention

Pure resource contention is modelled by making the *UpdateProb* zero, hence all transactions need to only read pages hence all of them can be given the lock to the pages. It can be seen from the Figure 6.3c that under pure resource contention, the *RestartRatio* is zero since there is no data contention at all.

6.5 Experiment 4: Parallel Transaction Execution

In the previous experiments, we considered *sequential* transaction execution. In this experiment, we consider *parallel* transaction execution in order to investigate the effect of parallel execution of cohorts on the relative real-time performance of the various commit protocols. We conducted this experiment for both RC+DC and pure DC scenarios. The parameter values under RC+DC and pure DC scenarios were the same as those used in Experiment 1 and Experiment 2, respectively—except *TransType*, which of course is "parallel" for this experiment. Figures 6.4a–6.4h present the results of this experiment for RC+DC scenario while Figures 6.5a–6.5h present the results for pure DC scenario.

From these figures we see that the observations made in the previous experiments for sequential transactions, to a large extent, hold true for the parallel transactions also. In particular, the effect of distributed commit processing on the MissPercent performance remains considerably more than that of distributed data processing [28]. The performance of PA is similar to that of 2PC, while the performance of PC is slightly worse than that of 2PC. The performance of 3PC is worse than that of 2PC, and RT-OPT continues to provide better performance as compared to 2PC.

For the RC+DC case (Figures 6.4a–6.4h), we observe that the performance differences

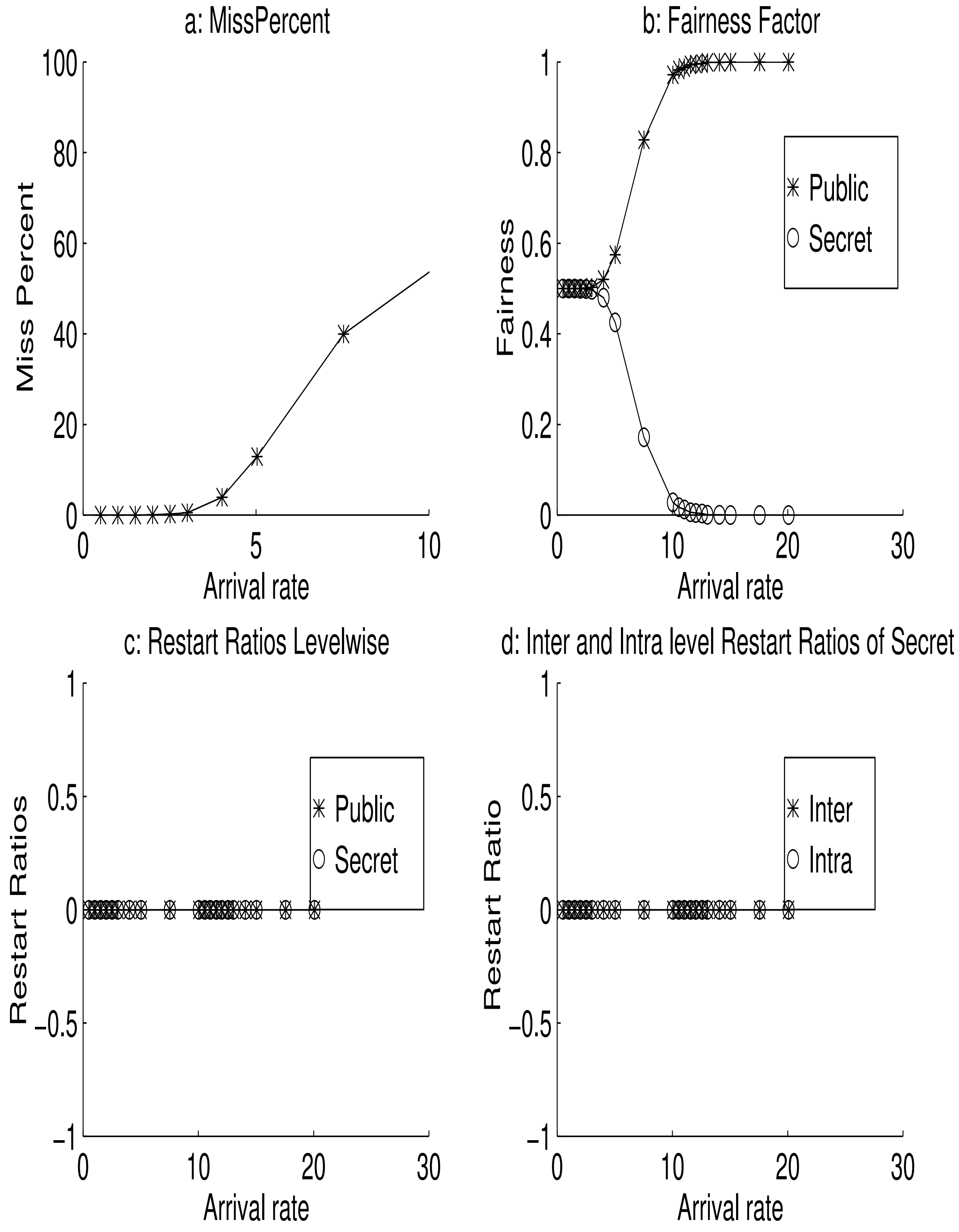

Figure 6.3: (a–d) Pure Resource Contention (Parallel)

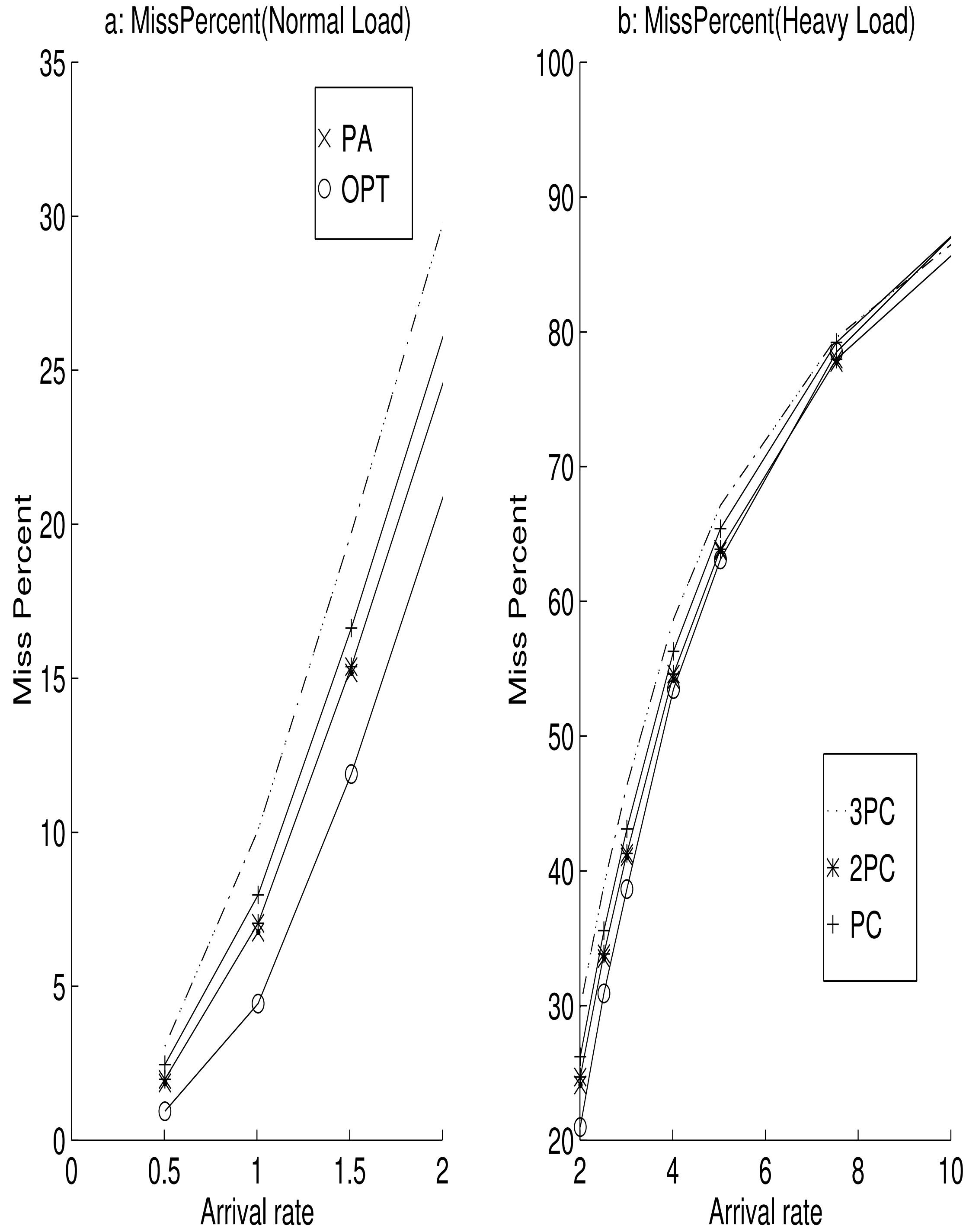

Figure 6.4: (a–b) Parallel Transactions (RC+DC)

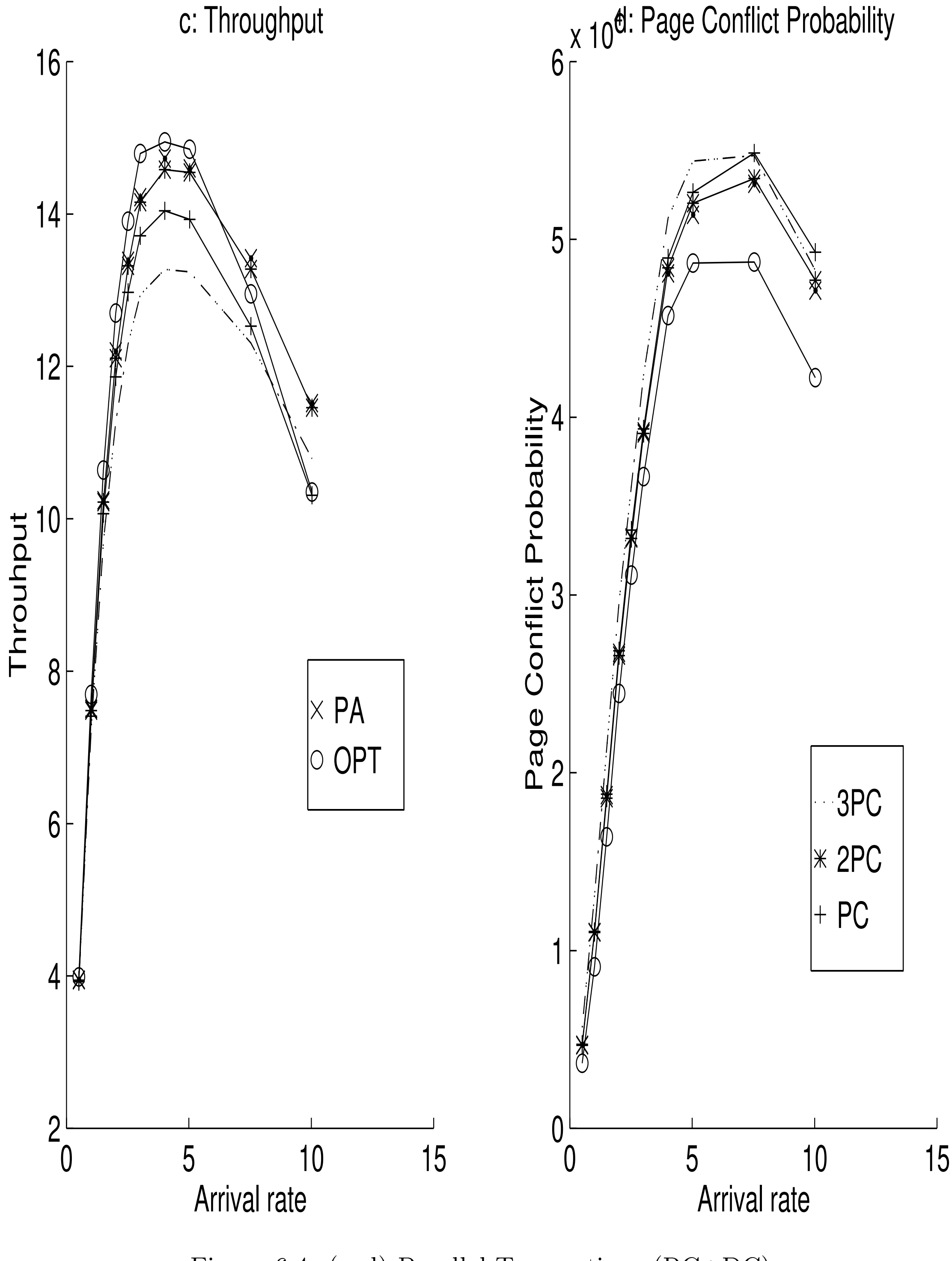

Figure 6.4: (c–d) Parallel Transactions (RC+DC)

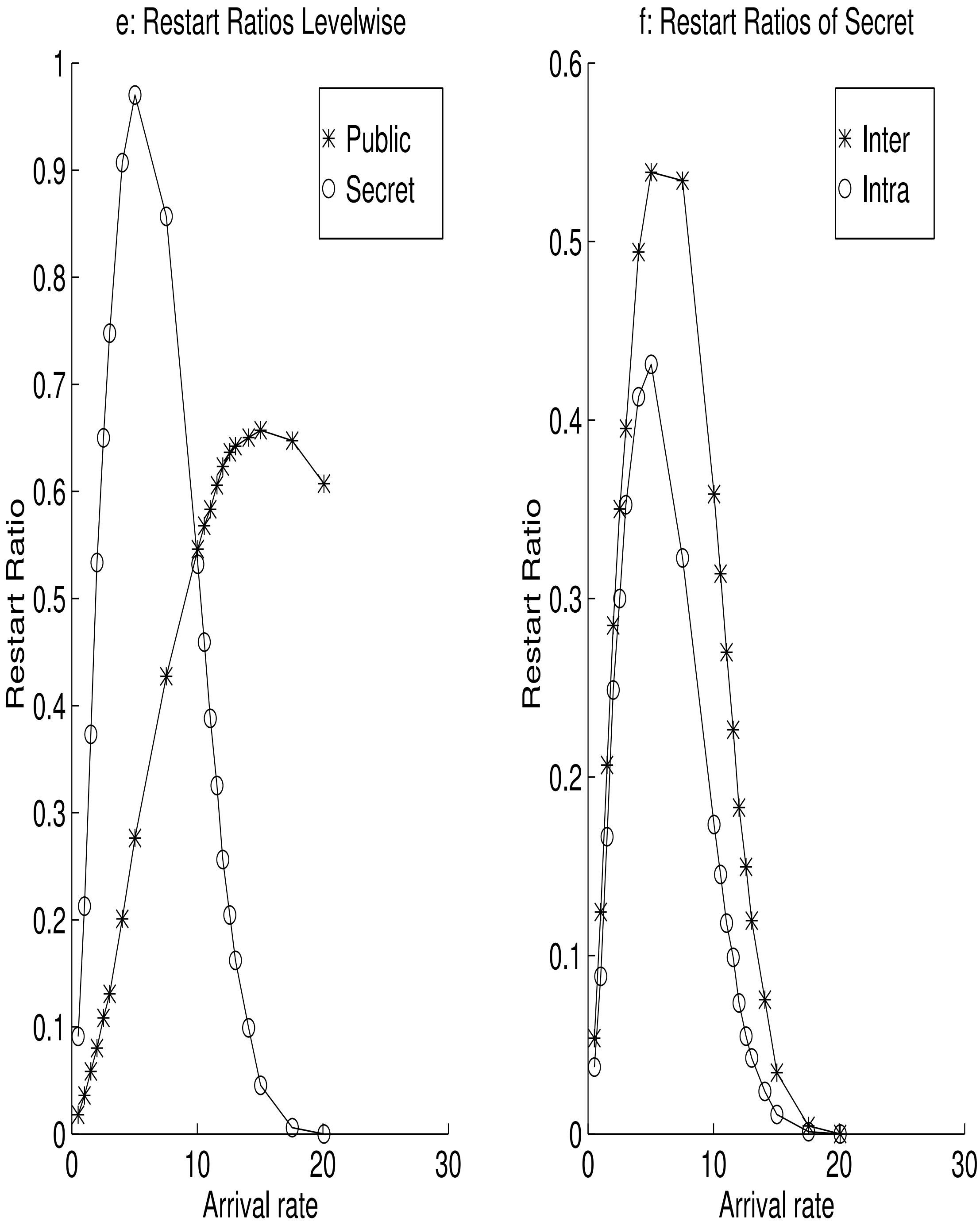

Figure 6.4: (e–f) Parallel Transactions (RC+DC)

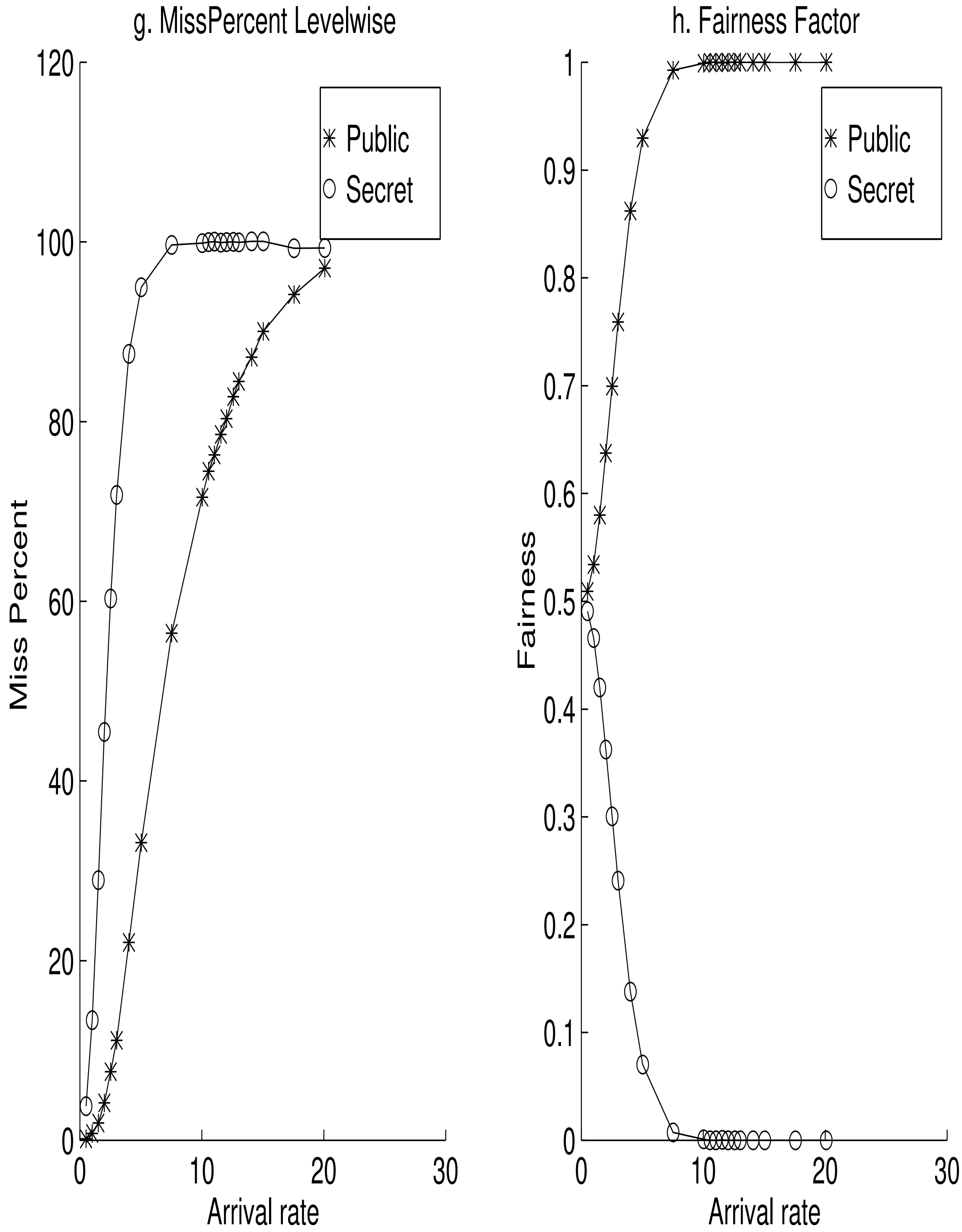

Figure 6.4: (g–h) Parallel Transactions (RC+DC)

between 2PC and 3PC have increased as compared to those seen for the sequential transaction execution experiments. The reason for this is that in parallel transaction execution, the cohorts execute in parallel, thus reducing the transaction response time, but the time required for the commit processing remains the same. The overall MissPercent is less in parallel execution than for the sequential transaction execution since the execution of the cohorts are in parallel. As the overheads of 3PC are more than that of 2PC, the effects of 3PC are felt to a greater extent.

Another observation from these figures is that the effect of RT-OPT on the MissPercent performance is reduced to some extent as compared to the sequential transaction execution. This is explained as follows: In sequential transaction execution, the benefits of RT-OPT are due to the optimistic feature and due to the active abort policy. In parallel transaction execution, however, the active abort policy does not improve the performance. This is because even if a cohort notifies the master about the abort, the chances are that the master has already sent the PREPARE messages to all the cohorts before receiving such a message. Moreover, because the cohorts execute in parallel, there are much less chances that a cohort may be aborted after sending the WORKDONE message, but before receiving the PREPARE message—that is where the active abort policy helps in the sequential transaction execution. Therefore, in the parallel case, there are no benefits due to the active abort policy, and hence the performance of RT-OPT is less impressive as compared to the sequential case.

Under pure DC scenario, however, the performance of RT-OPT is again comparable to that seen for the corresponding sequential transaction execution experiment. The reason for this is that under pure DC, as there is no resource contention, RT-OPT reduces the data contention significantly . Since in this case, the borrow ratio and the success ratio for RT-OPT are significantly high , thus making RT-OPT perform considerably better than the standard commit protocols.

Comments about the graphs for $Fairness$, $RestartRatios$ and $MissPercent$ are simmilar to that in the Sequential experiments.

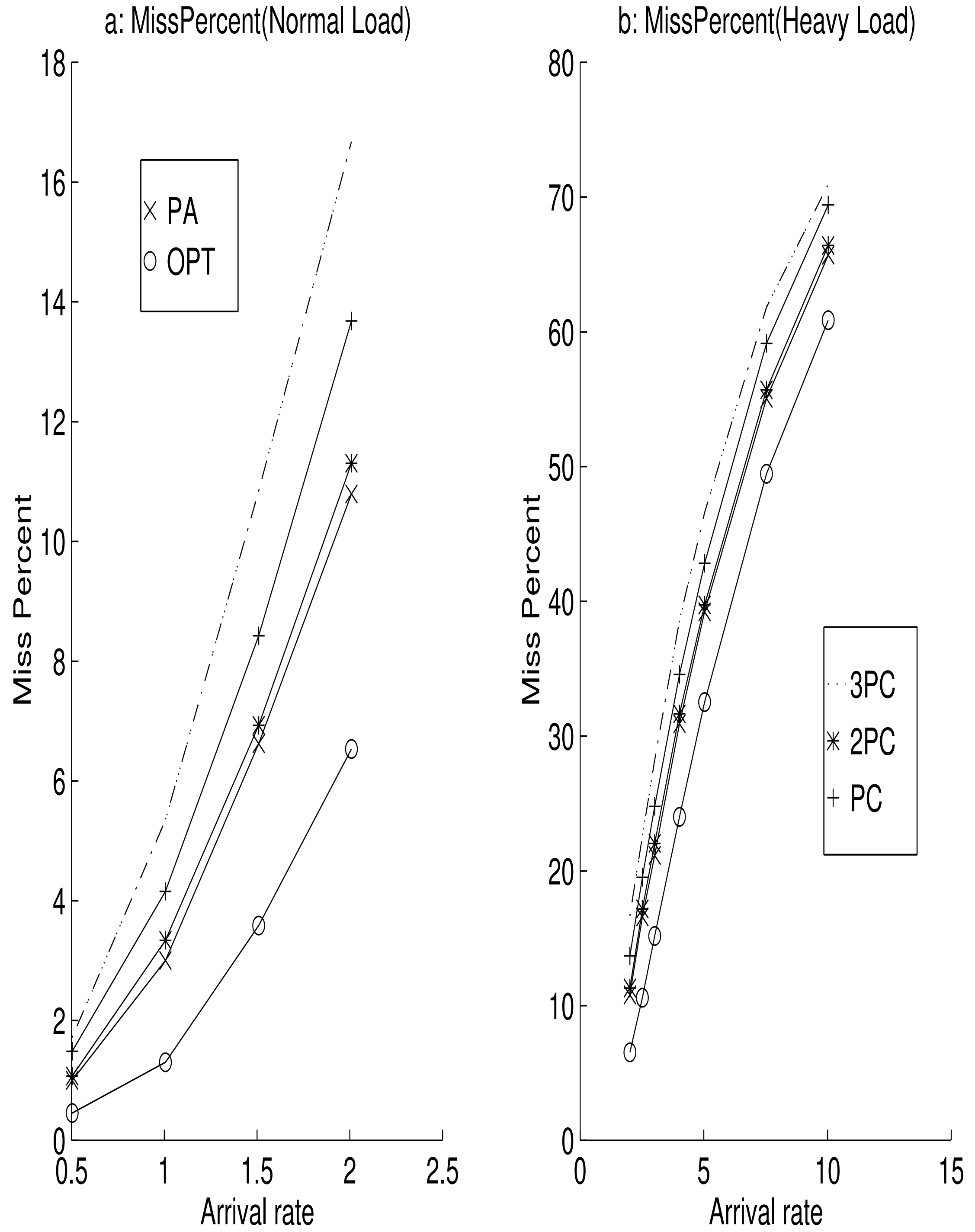

Figure 6.5: (a–b) Parallel Transactions (PureDC)

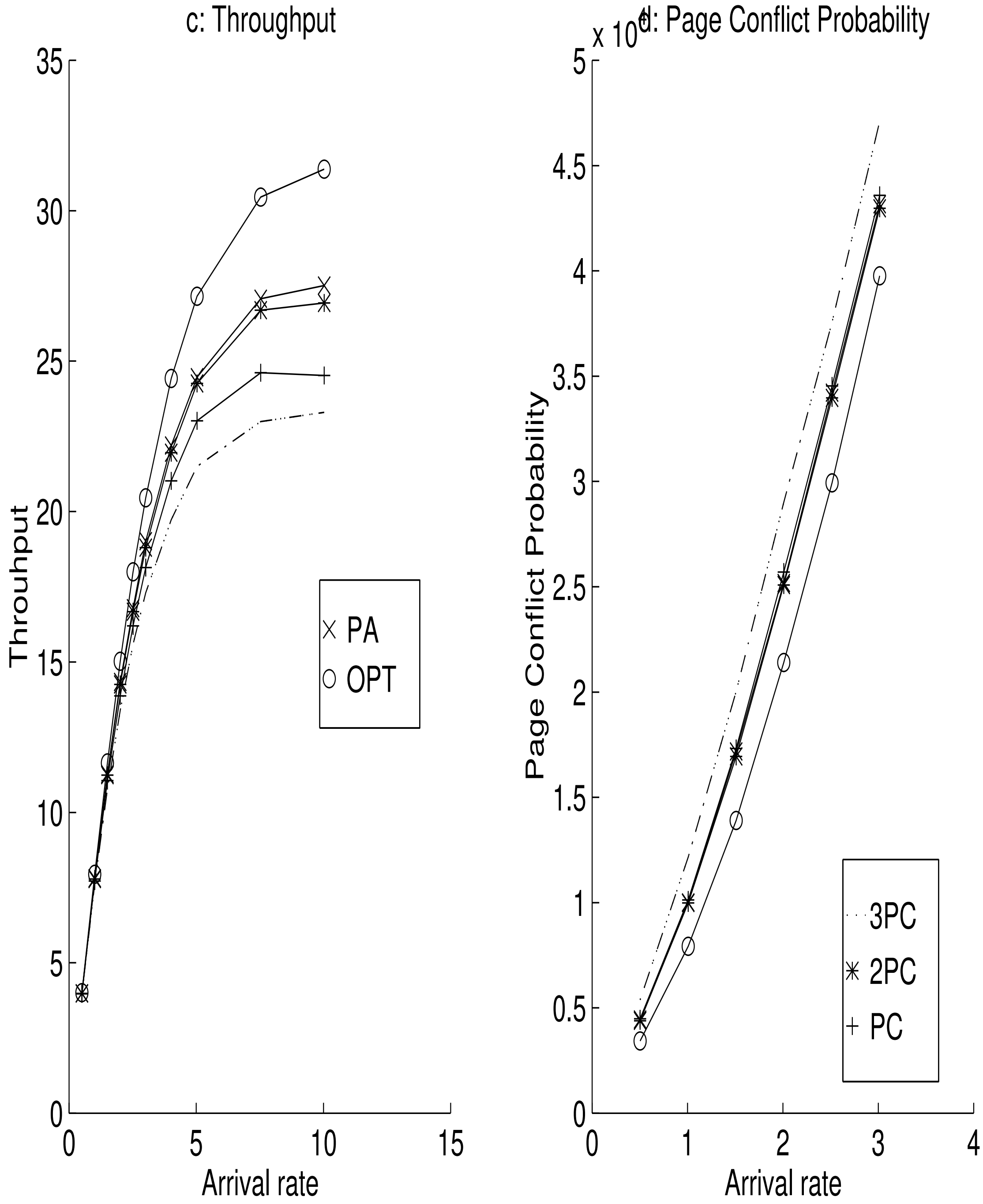

Figure 6.5: (c–d) Parallel Transactions (PureDC)

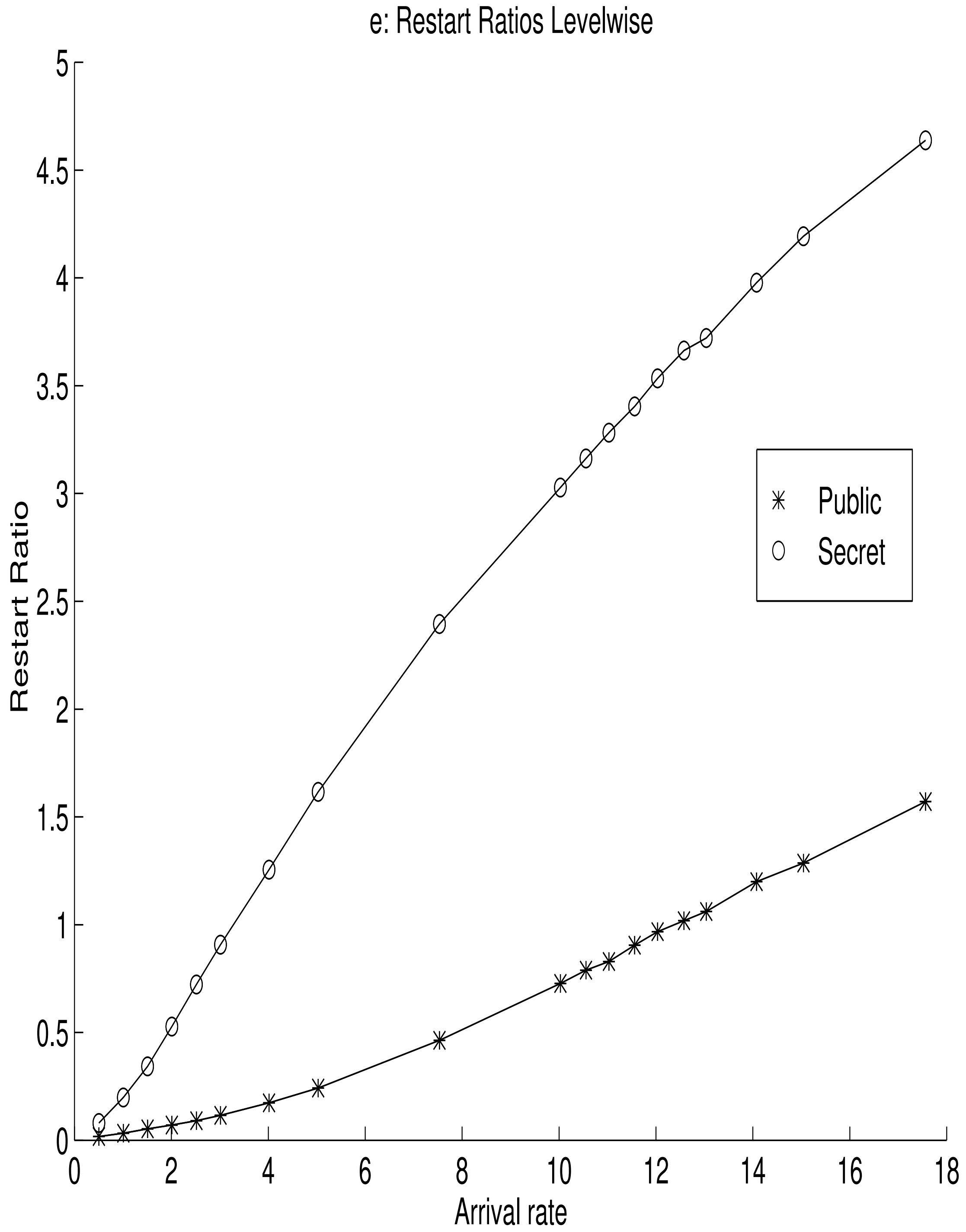

Figure 6.5: (e) Parallel Transactions (PureDC)

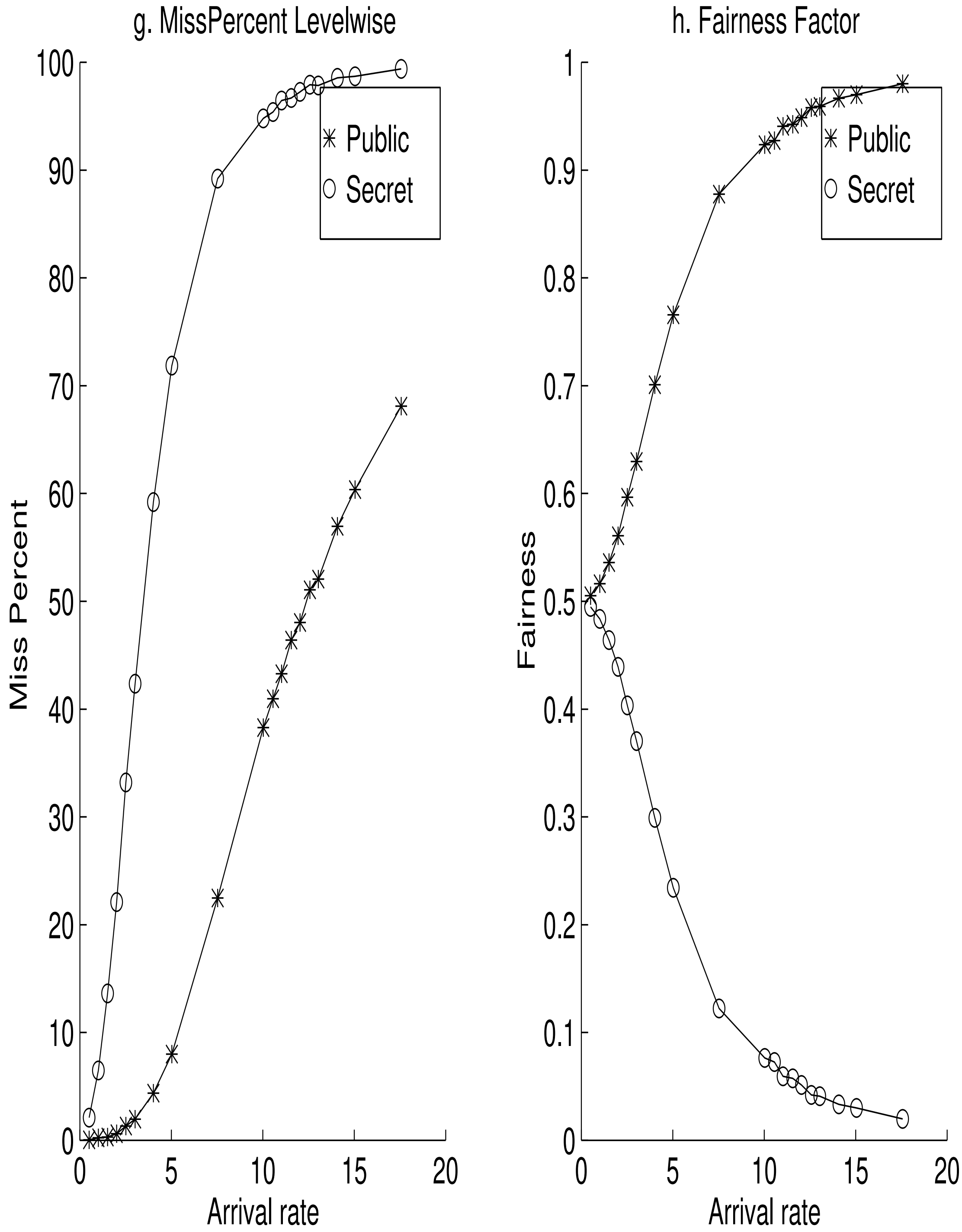

Figure 6.5: (g–h) Parallel Transactions (PureDC)

6.6 Experiment 5: Non-Blocking RT-OPT

This experiment shows the result of combining RT-OPT with the non-blocking 3PC protocol. It can be observed from the Figure 6.6a and Figure 6.6b that the OPT-3PC does better than 2PC and much better than 3PC, but ofcourse worse than OPT-2PC due to the price paid for non-blocking functionality.

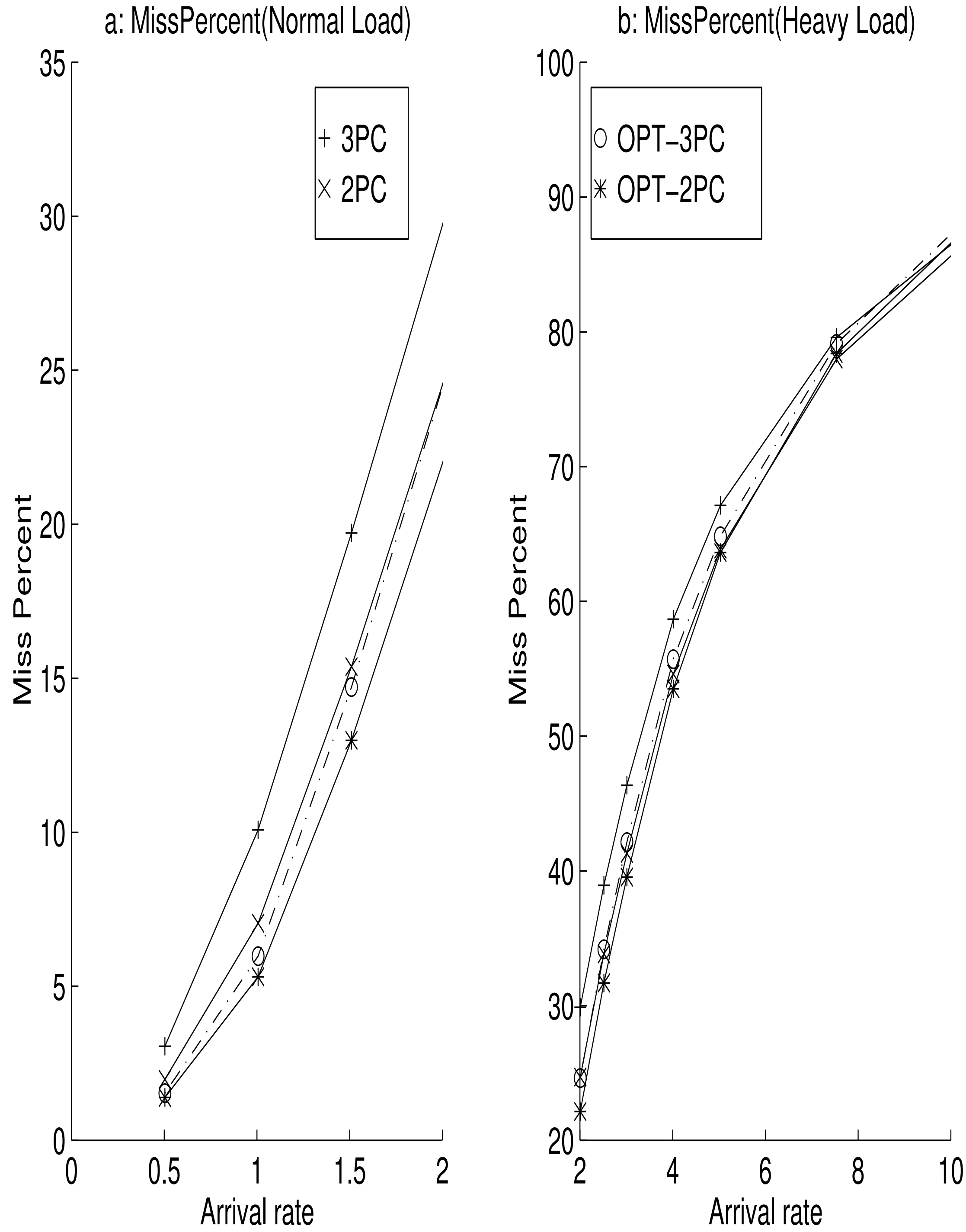

Figure 6.6: (a–d) Non-Blocking OPT (Miss Percent Parallel Transactions)

CHAPTER

7

Conclusions and Future Research

7.1 Summary of Contributions

In this thesis, we have quantitatively investigated the design of high performance transaction processing mechanisms for maintaining covert-channel-free security in real-time database systems. Such systems find application in safety-critical installations and military units where enforcing security is crucial to the success of the enterprise.

Apart from ensuring the data integrity constraints, a secure real-time database system has to simultaneously satisfy two requirements, namely, provide security and minimize the number of missed deadlines. Unfortunately, the mechanisms for achieving the individual goals often work at cross-purposes. Therefore, the design of secure transaction processing protocols involves considerable design complexity. Unlike previous studies, which used a *tradeoff* approach between security and timeliness, we have considered security for Full-secure applications for whom security is as an "all-or-nothing" issue. In comparison to these *correctness* criterion, the number of missed deadlines is a *performance* issue. Therefore, our study investigates the problem of how to minimize, as far as possible, the number of missed transaction deadlines *without* compromising security.

For fully secure applications, in order to satisfy the requirement of non-interference, only those real-time CC protocols that are free from priority inversion can be used. Hence 2PL-HP protocol has been used in our SRTDBS model. Although a significant body of research literature exists for centralized real-time database systems, comparatively little work has been done on distributed Secure RTDB systems. Using a detailed simulation model of a firm deadline Secure RTDB system, we evaluated the deadline miss performance of the real-time variants of a variety of standard commit protocols including 2PC, Presumed Abort, Presumed Commit, 3PC and RT-OPT. designed specifically for the real-time environment

We first precisely defined the process of transaction commitment and the conditions under which a transaction is said to miss its deadline in a firm deadline distributed real-time environment. Subsequently, we made a detailed study of the relative performance of different commit protocols in a firm deadline distributed real-time database system. Using a detailed simulation model of a firm deadline distributed RTDB system, we evaluated the deadline miss performance of the real-time variants of a variety of standard commit protocols including 2PC, Presumed Abort, Presumed Commit, and 3PC. We also evaluated the new commit protocol, RT-OPT [28], that was designed specifically for the real-time environment.

Our experiments, which covered a variety of transaction workloads and system configurations, demonstrated the following:

1. The standard 2PC and 3PC algorithms perform poorly in the real-time environment due to preventing access to data held by cohorts in the prepared state, and due to their passive nature.

2. The PA and PC variants of 2PC, although reducing protocol overheads, fail to provide tangible benefits in the real-time environment. These protocols provide improved performance in conventional database systems in certain situations, for example, higher degree of distribution. In real-time domain, however, not even a single experiment in our study exhibited more than marginal performance improvement of PA, while PC actually performs slightly worse than 2PC under many situations. However, this conclusion is limited to the *update-oriented* transaction workloads considered here.

3. The new protocol, RT-OPT, provides significantly improved performance over the

standard algorithms for almost all of the workloads and system configurations considered in this study. Its good performance is attained primarily due to its optimistic borrowing of uncommitted data and active abort policy. The optimistic access of the prepared data significantly reduces the effect of priority inversion which is inevitable in the prepared state and also avoids the covert channels formed during the prepared state. The other optimizations of silent kill and presumed commit/abort, however, had comparatively little beneficial effect.

4. Experiments combining the optimizations of RT-OPT with 3PC indicate that the non-blocking functionality can be obtained in the real-time environment at a relatively modest cost in normal processing performance. This is especially encouraging given the high desirability of the non-blocking feature in the real-time environment.

5. The performance of the RT-OPT protocol is, in some sense, close to the *optimal* performance.

7.2 Future Research

The research that we have presented in this thesis can be extended in a variety of ways:

- Our study was focused on transaction processing systems where transactions have “firm” deadlines. Some of our protocols may be extended to “soft” deadline systems, where transactions are allowed to continue even after the expiry of their deadlines.

- A basic assumption in our study is that the SRTDBS has no a-priori knowledge of transaction processing requirements or of transaction data accesses. It would be worthwhile to investigate how, and to what extent, performance could be improved if the SRTDBS were to be apriori provided with more information about transaction characteristics (e.g., transaction size estimates).

- Through these experiments we have observed that the *Secret* transactions suffer a greater miss percent due to the *Public* transactions being given higher priority, hence work has to be done in designing algorithms which will make sure that equal fairness is ensured for all the different levels of transactions.

In summary, our results have shown that the conventional commit protocols perform poorly in a real-time environment, the PA/PC optimizations fail to provide tangible benefits even though these optimizations are used in many commercial conventional database products, and the performance of RT-OPT in firm Secure deadline RTDB systems is considerably superior to that of the standard commit protocols.

References

[1] P. Amman, S. Jajodia and I. Ray, "Ensuring atomicity of multilevel transactions", *Proc. of IEEE Symp. on Security and Privacy*, Oakland, CA, 1996.

[2] V. Atluri, S. Jajodia, T. F. Keefe, C. MaCollum and R. Mukkamala, "Multilevel Secure Transaction Processing: Status and Prospects", *Database Security, X: Status and Prospects*, Chapman & Hall 1997, eds. Pierangela Samarati and Ravi Sandhu.

[3] B. T. Blaustein, S. Jajodia, C. D. McCollum and L. Notargiacomo, "A Model of atomicity for multilevel transactions", *Proc. of IEEE Symp. on Security and Privacy*, Oakland, California, 1993.

[4] B. George and J. Haritsa, "Secure Transaction Processing in Firm Real-Time Database Systems", *Proc. of ACM SIGMOD Conf.*, May 1997.

[5] D. Russel and G. T. Gangemi, *Computer Security Basics*, O'Reilly and Associates, Inc., 1991.

[6] J. Goguen and J. Meseguer, "Security Policy and Security Models", *Proc. of IEEE Symp. on Security and Privacy*, 1982.

[7] J. Haritsa, "Transaction Processing in Firm Real-Time Database Systems", *Ph.D. Thesis*, Computer Sciences Department, University of Wisconsin, Madison, 1991.

[8] J. Haritsa, M. Carey and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems", *Real-Time Systems Journal*, 4(3), 1992.

[9] T. F. Keefe and W. T. Tsai, "Multiversion concurrency control for multilevel secure database systems", *Proc. of IEEE Symp. on Security and Privacy*, Oakland, California, 1990.

[10] B. W. Kernighan and D. M. Ritchie, *C Programming Langauge* Prentice-Hall, 1988.

[11] C. J. Date, *AN INTRODUCTION TO DATABASE SYSTEMS*, Addison-Wesley Publishing Company, 1994.

[12] AHMED K ELMAGARMID, *DATABASE TRANSACTION MODELS For Advanced Applications*, MORGAN KAUFMANN PUBLISHERS.

[13] L. LaPadula and D. Bell, "Secure computer systems: Unified Exposition and Multics Interpretation", *The Mitre Corp.*, March 1976.

[14] M. C. little and D. L. McCue *Construction and Use of a Simulation Package in C++* Xerox Corporation, Webster, New York 14580

[15] C. Mohan, B. Lindsay and R. Obermarck, "Transaction Management in the R* Distributed Database Management System", *ACM Trans. on Database Systems*, 11(4), 1986.

[16] H. Pang, M. Carey and M. Livny, "Transaction Scheduling in Multiclass Real-Time Database Systems", *Proc. of IEEE Real-Time Systems Symp.*, December 1992.

[17] J. Peterson and A. Silberschatz, *Operating Systems Concepts*, Addison-Wesley, 1986.

[18] I. Ray, E. Bertino, S. Jajodia and L. Mancini, "An Advanced Commit Protocol for MLS Distributed Database Systems", *Proc. of 3rd Intl. Conf. on Computer and Communication Security*, ND, India, March 1996.

[19] A. Silberschatz, H. F. Korth and S. Sudarshan *Database System Concepts*, 3rd ed., McGraw-Hill, 1997.

[20] S. Son, R. David and B. Thuraisingham, "An Adaptive Policy for Improved Timeliness in Secure Database Systems", *Proc. of Annual IFIP WG 11.3 Conference of Database Security*, August 1995.

[21] S. Son, R. David and B. Thuraisingham, "Improving Timeliness in Real-Time Secure Database Systems", *ACM SIGMOD Record*, Special Issue on Real-Time Database Systems, March 1996.

[22] S. Son and B. Thuraisingham, "Towards a Multilevel Secure Database Management System for Real-Time Applications", *Proc. of IEEE Workshop on Real-Time Applications*, May 1993.

[23] Special Issue on Real-Time Database Systems, *Real-Time Systems Journal*, September 1992.

[24] J. Stankovic and W. Zhao, "On Real-Time Transactions", *ACM SIGMOD Record*, March 1988.

[25] O. Ulusoy, "Research Issues in Real-Time Database Systems", *Tech. Rep. BU-CEIS-94-32*, Department of Computer Engineering. and Information Science, Bilkent University, Turkey, 1994.

[26] Y. Yoon, "Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems", *Ph.D. Thesis*, Korea Advanced Institute of Science and Technology, May 1994.

[27] Ramesh Gupta, Jayant Haritsa and Krithi Ramamritham, "Revisiting Commit Processing in Distributed Database Systems", *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, USA, May 1997, *(to appear)*.

[28] Ramesh Gupta and Jayant Haritsa, "Commit Processing in Distributed Real-Time Database Systems", *Proceedings of the National Conference on Software for Real-Time Systems*, Cochin, India, January 1996, pages 195-204.

[29] R. Gupta, J. Haritsa, K. Ramamritham, and S. Seshadri, “Commit Processing in Distributed Real-Time Database Systems”, *Proc. of 17th IEEE Real-Time Systems Symp.*, Washington, DC, USA, Dec 1996.

[30] Ramesh Kumar Gupta, ”Commit Processing in Distributed On-Line and Real-Time Transaction Processing Systems”, *MSc Thesis*, Supercomputer Education and Research Centre, Indian Institute of Science, 1997.

[31] Binto George, ”Secure Real-Time Transaction Processing”, *Ph.D. Thesis*, Supercomputer Education and Research Centre, Indian Institute of Science, 1998.

[32] L. LaPadula and D. Bell, “Secure computer systems: Unified Exposition and Multics Interpretation”, *The Mitre Corp.*, March 1976.

[33] W. Lampson, “A Note on the Confinement Problem”, *Comm. of ACM*, October 1973.

[34] C. Mohan and Dick Dievendorff, “Recent Work on Distributed Commit Protocols, and Recoverable Messaging and Queuing”, *Data Engineering,vol 17 no 1,1994*

[35] C. Mohan and B. Lindsay and Ron Obermarck, “Transaction Management in the $R^\star$ Distributed Database Managem ent System”, *Transactions on Database Systems, vol 11 no 4,1986*

[36] T. Keefe, W. Tsai and J. Srivastava, “Multilevel Secure Database Concurrency Control”, *Proc. of IEEE Intl. Conf. on Data Engineering*, February 1990.

[37] Jim Gray, ”Personal communication”, *undo by invalidation of buffer pages aug 1996*

[38] B. Lampson and H. Sturgis, ”Crash Recovery in a Distributed Data Storage System”, *Xerox, Palo Alto Research Center*

[39] Jim Gray, ”Notes on Database Operating Systems”, *60, Springer-Verlag, Berlin*

[40] Rakesh Agrawal and David J. Dewitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", *Transactions On Database Systems*

[41] C. Mohan and Don Haderle and Bruce Lindsay and Hamid Pirahesh and Peter Schwarz, *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*

[42] Theo Haerder and Andreas Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Computing Surveys.*

[43] Jim Gray and Andreas Reuter, "Transaction Processing: Concepts and Techniques", *Morgan Kaufmann.*

[44] D. G. Severance and G. M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases", *Transactions On Database Systems*