The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

PERFORMANCE ASPECTS OF SECURITY-AWARE DATABASE SYSTEMS

A Thesis in

Computer Science and Engineering

by

Sudsanguan Ngamsuriyaroj

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2002

We approve the thesis of Sudsanguan Ngamsuriyaroj.

Date of Signature

Thomas F. Keefe
Oracle Corporation
Thesis Co-Adviser
Co-Chair of Committee
Special Member

Ali R. Hurson
Professor of Computer Science and Engineering
Thesis Co-Adviser
Co-Chair of Committee

John Hannan
Associate Professor of Computer Science and Engineering

John J. Metzner
Professor of Computer Science and Engineering
and Electrical Engineering

Joseph M. Lambert
Associate Professor of School of Information Science and Technology
and Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

Abstract

Security as well as performance are crucial requirements in every application design. However, security requirements of an application impose some limitations and typically have a significant impact on the performance of the application. In this thesis, we address security issues in distributed database systems, propose solutions to impose security in underlying platforms, and study the performance implications of our proposed solutions. The objective is to design efficient solutions for distributed database systems that satisfy security requirements while offering acceptable performance.

We consider three database paradigms: multilevel databases, replicated databases, and multidatabases. Each paradigm has its own unique security requirements. In a multilevel database, every component of the database is run under a multilevel security policy. Any accesses to the database must be free of potential covert channels. In this work, a multilevel log manager is designed and implemented to support logging of multiple security levels. The design uses a round-robin approach to log transaction activities of each security level. The design is channel-free and also offers good performance.

Maintaining the consistency of a common security policy in a distributed environment can be modeled as a replicated database. Strong consistency such as one-copy serializability is required since any inconsistency including transient inconsistency may lead to a security violation. Invalidation-based consistency protocol proposed in this thesis satisfies one-copy serializability and performs better than update-based protocols.

Heterogeneity and local autonomy are main characteristics of multidatabases. Summary Schemas Model (SSM) is an adjunct to a multidatabase. It helps resolve name and semantic heterogeneity in the multidatabase by defining access terms as hypernyms or hyponyms from data names of underlying local databases. As a result, accessing heterogenous data in multidatabases via access terms defined in the SSM is simple and efficient. However, those terms are publicly accessible and unprotected. Thus, an SSM authorization model is presented. The model restricts access to SSM access terms according to global roles defined at multidatabase level. Consequently, unauthorized accesses are rejected before they reach local databases. This reduces the network traffic, decreases the workload at database servers, and hence offers higher performance.

In conclusion, we present three topics to illustrate that, even under security requirements of database systems, the underlying environments are able to exhibit good performance. We believe that the ideas presented here will contribute to the design of secure and efficient database environments.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Abbreviation	Referenced Terms	Pages
1SR	One-Copy Serializability	17
2PL	Two-Phase Locking	11
ACL	Access Control List	24
CPL	Capability List	24
DAC	Discretionary Access Control	24
DBMS	Database Management System	30
DTOS	Distributed Trusted Operating System	31
IPC	Interprocess Communication	36
ILBP	Invalidation Lock-Based Consistency Protocol	70
ITBP	Invalidation Timestamp-Based Consistency Protocol	70
LBC	Level Buffer Chain	41
LGM	Log Manager	30
LSN	Log Sequence Number	32
MAC	Mandatory Access Control	24
MDBS	Multidatabase System	19
MLS	Multilevel Security	25
RM	Resource Manager	30
SDM	Semantic Distance Metric	21
SG	Serializability Graph	15
SID	Security Identifier	36
SSM	Semantic Schema Model	20
STAR-DBS	Secure TransActional Resources Database System	31
TM	Transaction Manager	30
WADS	Write-Ahead Data Set	34
WAL	Write-Ahead Logging	32

Acknowledgments

First of all, I am grateful to both of my thesis advisors, Dr. Thomas F. Keefe and Dr. Ali R. Hurson, for their guidance, patience, and encouragement during my research at Penn State. I would not be able to finish this work without their help. Thanks also go to Dr. John Hannan, Dr. John Metzner, Dr. Joseph Lambert for their valuable comments and for serving on my thesis committee.

I am thankful to the Government of Thailand and to the Department of Computer Science, Mahidol University for their financial support. I also would like to thank the Department of Computer Science and Engineering at Penn State for supporting me as a teaching assistant. I am grateful to Dr. Supachai Tangwongsan who has been genuinely supporting me for pursuing this degree. I appreciate Dr. Damras Wongsawang for his support.

During my study here, I have often received help and encouragement from a number of my Thai friends I just made in State College. My special thanks go to Dr. Chalie and Srijidtra (Taew) Charoenlarpnopparat for their hospitality when I often stopped by for delicious meals and good movies. My sincere thanks also go to Dr. Nandh Thavarungkul for helping me get settled during my early years here. I would like to thank Nuntawan (Lynn) Silpngarmlers and Tanes Ruangturakij for their helps. I also would like to thank Dr. Rattikorn and Wipa Yimnirun, Rattana Tantatherdtam, Atitsa Petchsuk and Mantana Kanchanasopa for making my recent time here very enjoyable. I thank Dr. Sun-Euy Kim for being my best friend in the Department. I truly thank Vicki

Keller for her help and support. My special thanks also go to Nuchanart Sooppipatt, my best friend back in Thailand, for her support even though we are very far away. My best thanks go to my long-time best friends for more than 15 years, Nuntawan (Noy) Intaravitak and Nantiya Nash, whose continuous encouragement, support, help, and listening have always lit up my spirit when things were down.

My greatest thanks go to my family, especially my sisters, Chanpen Pansin and Patcharin Ngamsuriyaroj for their love and support. They have taken good care of my matters in Thailand during my study here. Finally, I would like to dedicate this thesis to my late parents for their unconditional love and for raising me to be a person as I am now.

Chapter 1

Introduction

In a typical application, various design choices are generally suggested so that the best possible performance can be achieved. However, in designing a secure application, security requirements limit the available number of design choices and hence, in many cases, good performance would have to be compromised. In other words, in many instances, the most efficient design may not satisfy security requirements of the application.

Security requirements for an application usually include restrictions to data accesses, and limitations of processing capabilities according to control rules specified by a security policy. For instance, only a set of privileged users can access a specific set of sensitive data. There are various security aspects to be considered in a particular application depending on (i) what vulnerability or weakness the application might contain that can be exploited to cause loss or harm, and (ii) what potential threats the application may receive. The goals of security enforcement are *confidentiality*, *integrity*, and *availability* [40]. Confidentiality means that only authorized persons can access protected data. Integrity refers to the ability to modify data by only authorized persons or only in authorized ways. Availability means that data is accessible and should not prevent any legitimate access, or so-called *denial of services* should be prevented.

A database is a collection of related organized data managed by a database management system. Database systems have become an essential part of an application [40]. They are intended to be shared among several users to minimize redundancy and to ensure data consistency. Like other sharing systems, basic security requirements of a database system may include access control, integrity, user authentication, inference, auditability, and availability. Note that auditability refers to the ability to trace who access the database and inference refers to the ability to access sensitive data *indirectly*.

Database systems may be categorized as distributed databases, replicated databases, multilevel databases and multidatabases. Each database paradigm may have different degree of individual security requirements. For example, multilevel databases emphasize preventing inference due to potential unintended and indirected communication, replicated databases have a concern on data consistency and integrity in addition to availability, and multidatabases have to deal with heterogeneous access control of local databases.

Security and performance usually have conflict interest. This is because adding security enforcement often increases the size and complexity of a database system. In this thesis, we study the effect of security requirements on the performance of applications running within selected database environments.

1.1 Motivation

Performance is always a major concern when designing an application. Security requirements imposed on the application have a significant impact on its performance because of extra overhead and more restricted access to data determined by the security

policy of the system. This thesis is intended to evaluate the performance of applications running on selected database platforms when security requirements are imposed.

We consider security requirements in three database systems: multilevel databases, replicated databases, and multidatabases since each database system has its own unique security requirements which affect both the design choices and the performance of its applications.

In multilevel databases, transactions of every security level are executed under multilevel security policy which requires that transactions running at one level must not be interfered by those running at another level [14]. As a result, all database components designed and implemented for use in a typical database system must be modified to ensure that there is no indirect interference of transactions executed among security levels.

A log manager is an important part of any database system since it maintains logs of transaction activities so that any transaction can be recovered when there is a failure. Logs of transaction activities must be persistently recorded on disk to guarantee recovery of those transactions. Obviously, the performance of the log manager greatly affects the overall database performance. The addition of multilevel security puts even more restrictions on log management since an interference can occur when the log manager has to manage logs from multiple security levels at the same time. As a result, the log manager may not perform well under multilevel security requirements. Our motivation is to design a multilevel log manager that achieves good performance.

A replicated database can be used to model a common and consistent security policy replicated in a distributed environment [34, 35]. Unlike replicated data which

may be allowed to be inconsistent for a period of time, the security policy must be consistent at all times to prevent any security violation. In other words, the standard for having such consistency is one-copy serializability since it gives a one-copy view of the policy regardless of which replica is accessed. Consistency protocols normally used for maintaining data consistency may not be suitably applied. Even though update-based consistency protocols such as eager replication ensure one-copy serializability, they often have poor performance. Thus, our motivation is to investigate invalidation-based consistency protocols that are simple to implement, generally outperform update-based consistency protocols, and guarantee one-copy serializability.

A multidatabase system (MDBS) integrates existing and possibly heterogeneous databases while preserving individual local autonomy. It represents individual local databases globally and allows distributed accesses across them. However, each local database may have heterogeneous data models and query languages. In addition, each individual local database is autonomous and has complete control over its data and resources. It can join or depart the MDBS at any time without major changes. The MDBS must be able to form a common global data model and resolve data name or semantic differences with no significant impact to local operations [44].

Creating a logical global view of data by integrating individual heterogeneous local schemas is not an easy task. Similarly, forming a global authorization for the MDBS by integrating local authorizations of individual local databases is quite a challenge and even more complicated because (i) authorization models of local databases are possibly in conflict with each other due to heterogeneity, and (ii) their authorization autonomy

cannot be compromised. In addition, subjects and objects among local databases are very likely incompatible.

Summary Schemas Model (SSM) proposed in [11] provides an efficient means to access heterogeneous forms of data in the MDBS. The model resolves data naming and semantics heterogeneity using word relationships defined in a standard dictionary. The model builds a hierarchy structure of metadata based on access terms exported from local databases. As a result, global accesses to those terms are feasible without knowing their precise name and location, and without modifying any local schema.

For security concerns, the model does not have any protection at its metadata hierarchy structure. Thus, it allows public accesses to those terms. Even though the legitimacy for accessing the terms can be examined and authorized by local authorization, the model is still vulnerable for denial-of-service attacks. For example, if the hierarchy of the model is full of illegitimate accesses, it will take a long time before one legal access can get in.

The motivation of our work is to propose a global authorization for the SSM model so that access to its metadata can be restricted at the multidatabase level and unauthorized accesses can be detected before they reach local databases. The global authorization proposed should accommodate the hierarchical nature of the SSM model for resolving authorization heterogeneity of local databases while respecting their authorization autonomy. Eventually, the performance of the proposed global authorization is evaluated.

1.2 Overview of the Thesis

In this thesis, we study performance issues of three database platforms under restrictions imposed by a security policy. The main goal is to provide solutions that not only satisfy security requirements, but also give good performance.

- We study a design and implementation of a multilevel log manager as part of a secure multilevel database system running under a secure operating system. We also study the impact of multilevel security requirements on the performance of the design.

- We study invalidation-based protocols for maintaining consistency of a common security policy replicated on several machines. The system we consider can be modeled as a replicated database. The protocols guarantee one-copy serializability and do not allow transient inconsistency among replicas of the policy since any inconsistency may lead to security violation. We evaluate the performance of the protocols and compare with eager replication; an update-based consistency protocol that also ensures one-copy serializability.

- We study and analyze an authorization model for the Summary Schemas Model to restrict accesses to its metadata. The authorization model proposed is based on role-based security policy since the policy can reflect the line of authority in a role hierarchy and, hence, it fits naturally with the hierarchical structure of the SSM. We also evaluate the performance of the proposed authorization model.

The content of this thesis is organized as follows. Chapter 2 covers necessary background on fundamental concepts in transaction processing on which most database system operations are based. Such concepts are concurrency control, consistency, and recovery. We also examine basic concepts of multidatabases and Summary Schemas Model. In addition, general ideas of security policy and multilevel security are introduced. Chapter 3 describes our design and implementation of a multilevel log manager. Chapter 4 studies two invalidation-based consistency protocols proposed. Chapter 5 proposes an authorization model for the Summary Schemas Model. Finally, Chapter 6 concludes the thesis, and discusses future research directions.

Chapter 2

Background

Our research involves three different kind of databases: multilevel databases, replicated databases, and multidatabases. The main interest is to study the system performance when security requirements are imposed on these database organizations. In the design of a database system, transaction processing plays a significant role because it constitutes essential mechanisms to ensure the correctness of the execution of any transaction in the system. Such mechanisms are concurrency control, recovery, and consistency protocols. Each mechanism contributes a direct effect on the performance of a database system. In addition, the imposition of security requirements in a database may force some changes in the design and consequently may affect the overall performance.

In this chapter, we describe those related topics in detail. First, we describe the concept of transaction processing and its main mechanisms: concurrency control, recovery, and consistency. Second, we explain consistency protocols to maintain data integrity in replicated, distributed database systems. The primary reason is that the protocols have a significant impact on the performance of accessing replicated data. To prevent security violations, authorization information for replicated data in a distributed system must be replicated and has to be consistent as well. Third, we introduce a multidatabase system and its main characteristics such as autonomy and heterogeneity. Due to these characteristics, ensuring the correctness of transaction execution in

multidatabases is rather complicated. Within the scope of multidatabases, we describe the so-called Summary Schemas Model (SSM) which resolves the heterogeneity of data naming in multidatabases. Subsequently, we will study the implication of security mechanism in the SSM and its impact on the performance of the model. Finally, we introduce some security policies such as multilevel security as they have unique requirements that affect the design and the performance of a database system.

2.1 Transaction Processing

In this section, we introduce transaction processing and its essential properties that define the correctness of a database system. We also discuss its fundamental mechanisms, concurrency control, consistency and recovery.

A transaction is a basic execution unit in a database system. It consists of a sequence of read and write operations performed on data items. In a typical database system, several transactions are executing simultaneously in order to achieve high throughput and fast response time. The execution of reads and writes of different transactions may be interleaved in many possible ways. However, the execution of two concurrent operations (except for two reads) on the same data in an uncontrolled manner may result in inconsistency due to the following update anomalies.

- **Lost Update**: If two transactions write to the same data item, one of the two updates may be lost.

- **Dirty Read**: If one transaction reads a data item written by another transaction which has not completed and committed yet, the value of the data item read has not been finalized and may be changed further.

- **Unrepeatable Read**: If one transaction reads a data item twice and another transaction changes the value of the data item between the two reads, the transaction reads two different values of the same data item.

2.1.1 ACID Properties

To ensure the correctness of the execution of a transaction, the transaction must satisfy the ACID properties [23]. They are:

1. **Atomicity**: A transaction is atomic in a way that all or none of its operations are performed.

2. **Consistency**: A transaction is consistent if its completion takes the database from one consistent state to another.

3. **Isolation**: A transaction is executed as if it is isolated from other concurrent transactions. In other words, its execution should not be interfered by other transactions.

4. **Durability**: A change made to the database by a committed transaction must be persistent regardless of any failure.

These properties are mainly enforced by concurrency control and recovery methods. Concurrency control helps prevent the three update anomalies by enforcing isolation

of transactions. Recovery techniques ensure that all changes committed to the database are persistent since the execution of a transaction may not be complete due to some failure such as power failure or user interruption.

2.1.2 Concurrency Control

Interference of interleaved execution of concurrent transactions can be avoided by enforcing a serial execution of transactions. In other words, interleaved operations of any pair of transactions are executed in such a way that they produce the same output and have the same effect on the database as though they are sequentially executed. However, for a set of transactions, there may be more than one sequence of serial executions and hence they may not produce the same result. To ensure the recoverability of a transaction, the system also enforces *strict* execution requiring that a transaction reading and writing to a data item x must be delayed until the transaction that previously wrote to x commits or aborts.

Two commonly used concurrency control techniques [6] to ensure the isolation of executing transactions are two-phase locking and timestamp ordering as described below.

2.1.2.1 Two-Phase Locking

Locking is a commonly used mechanism for synchronizing accesses to shared data. As the name indicated, two-phase locking (2PL) consists of two phases. The first phase is called *expanding* phase during which new locks can be acquired but none can be released. The second phase is called *shrinking* phase during which holding locks are released but

no new locks can be acquired. For strict execution, strict two-phase locking additionally requires that all locks held by a transaction are released only after the transaction commits or aborts. During lock holding by a transaction, no other transactions can get accesses to the data. As a result, the isolation of transactions is enforced. An adverse effect of locking is that it is subject to deadlocks. A deadlock can be avoided or prevented using a timeout or it can be detected using a *wait-for-graph* [47]. A transaction, however, may be aborted and restarted to resolve a deadlock.

2.1.2.2 Timestamp Ordering

In this technique, each transaction, T_i, is assigned a unique and totally ordered timestamp, $ts(T_i)$. In effect, every read and write in the transaction is tagged with $ts(T_i)$. Also, each data item x in the database is tagged with two timestamps: max_read_timestamp ($max\_r\_ts[x]$) which is the timestamp of latest read to x, and max_write_timestamp ($max\_w\_ts[x]$) which is the timestamp of the latest write to x. Conflicting operations (read-write or write-write) accessing x are scheduled using the following rules.

- A read operation, $r_i[x]$, is scheduled to read x if $ts(T_i) \geq max\_w\_ts[x]$. Otherwise, the read is rejected. $max\_r\_ts[x]$ is updated only if $ts(T_i) > max\_r\_ts[x]$.

- A write operation, $w_i[x]$, is scheduled to write if $ts(T_i) \geq max\_r\_ts[x]$ and $ts(T_i) \geq max\_w\_ts[x]$. Otherwise, the write is rejected. $max\_w\_ts[x]$ is updated only if $ts(T_i) > max\_w\_ts[x]$.

Any transaction whose operation is rejected will abort and may restart with a new and higher timestamp. Strict execution can be enforced by delaying any read or write to a data item x (with a higher timestamp than $max\_w\_ts[x]$) only after the transaction writing to x commits or aborts.

2.1.3 Recovery

Three common types of failures are transaction failures, system failures, and media failures. We focus only on recovery techniques due to transaction failures. In this case, the database is recovered when it can be restored to the most recent consistent state before the failure.

For recovery purposes, the system must know when a transaction begins, terminates, and commits or aborts. The system must maintain a log keeping track of all transaction activities that affect any change made to the database. The log is kept on disk and typically duplexed to guard against a single disk failure. Logically, the log is a sequence of log records containing information about data values written by transactions performing updates. A log record consists of two parts: the header and the body. The header contains a log record identification such as *transaction id* which tells who wrote the record and *log record id* known as log sequence number (LSN) which tells where the log record starts in the log. The body of a log record contains old and new values of data being updated so that either old values can be undone or new values can be redone.

Depending on a relative order of transaction commitment and writing updated data to the database on disk, a recovery process may require an undo or a redo or both [6]. An undo is required if an uncommitted transaction is allowed to write updates to

the database on disk. To restore the database to a committed state, any uncommitted updates have to be undone. A redo is required if a transaction is allowed to commit before its updates are written to the database on disk. Thus, before the transaction commits, its updates must be on disk, either a log file or the database itself.

Two main recovery techniques are *deferred update* and *immediate update* [18]. Deferred update delays writing updates of a transaction to the database on disk until after its commit point. Each update is written to the log before the transaction commits. If a transaction fails before it commits, there is no need to be undone. But, if a transaction fails after it commits but before its updates are recorded to the database on disk, all operations must be redone using log records in the log so that the effect of the committed transaction can be permanently recorded on the database. This is known as *no-undo/redo* approach.

In immediate update, updates are written to the database after they had been written to the log on disk without waiting for a transaction to commit. If a transaction fails before it commits, updates must be undone, but no redo is needed. This is called *undo/no-redo* approach. However, if a transaction is allowed to commit before all changes are written to the database, then a redo may be needed. This is called the *undo/redo* approach. To ensure that any update can be undone or redone, most database implementations use *write-ahead logging* (WAL) protocol. The protocol requires that every update in a transaction be written to the log on disk before it is recorded to the database.

2.1.4 Consistency and Serializability

Consistency can be achieved if transactions are executed in a serial fashion. Even though transactions can execute in any order, scheduling of operations in transactions must ensure serializability.

A history is a schedule of operations in transactions. It indicates a relative order of executed operations of transactions in which, for each transaction T_i, the order of its operations is the same as the order in the history. For n transactions, there are $n!$ possible serial histories.

Two histories are *view equivalent* [6] if they produce the same effect. In other words, two histories h and h' are view equivalent if (i) they contain the same set of transactions and the same operations; (ii) for any read, if T_i reads x from T_j in h, T_i also reads x from T_j in h'; and (iii) for each x, if $write_i[x]$ of T_i is the final write of x in h, then it is also the final write of x in h'. Also, two operations in a history are said to *conflict* if both access the same data and one of them is *write*. Two histories are *conflict equivalent* if the order of any two conflicting operations is the same in both histories. A history is *conflict serializable* if it is conflict equivalent to some serial executions of the same transactions.

Serializability theory [6] can be used to determine whether a history is serializable. Basically, the theory states that if a directed graph derived from a history H is acyclic, the history H is serializable. The graph called a *serializability graph* (SG) consists of all committed transactions T in the history H as nodes and all $T_i \rightarrow T_j$ (i $\neq$ j) as edges

where one of T_i's operations precedes and conflicts with one of T_j's operations in H. For example,

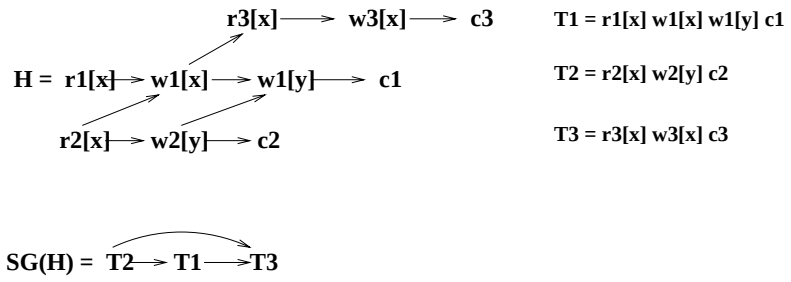

Fig. 2.1. A History and its Serializability Graph [6]

Note that a single edge can represent more than one pair of conflicting operations.

2.2 Consistency Protocols for Replicated Data

In the distributed environment, data can be replicated at several sites in order to enhance availability and improve performance. However, these advantages come at the expense of maintaining consistency across several replicas. In a one-copy database, an interleaved execution of a set of transactions should be equivalent to a serial execution of the same set of transactions. When data are replicated at many places, an interleaved execution of a set of transactions should be equivalent to a serial execution of those transactions on a one-copy database. Such execution is called one-copy serializable (1SR). In other words, the effect of data replication is transparent to those transactions.

Hence, the goal of consistency for replicated data is one-copy serializability (1SR) since it gives a one-copy view of replicated data regardless of which copy is accessed. However, maintaining 1SR consistency often leads to poor performance. As a result, in some applications, replicas are allowed to be inconsistent for a period of time. This consistency is called weak consistency [24] or transient consistency [41].

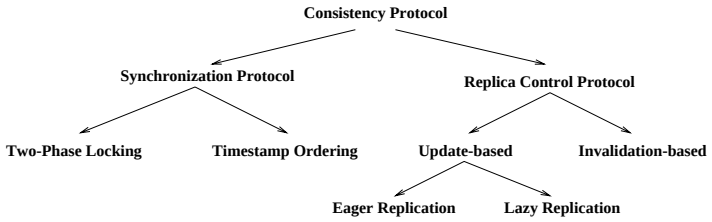

Fig. 2.2. Components of Consistency Protocols

As shown in Figure 2.2, a consistency protocol is composed of two parts. The first part is a synchronization protocol such as two-phase locking and timestamp ordering for controlling concurrent accesses to replicated data. The second part is a replica control protocol for controlling how changes at one site are propagated to other sites.

Two categories of replica control protocols are update-based and invalidation-based propagation [34]. In update-based propagation, an update is sent from the primary-copy site to other sites before or after it commits. In invalidation-based propagation, all copies are invalidated before the update commits. A new copy may be requested on demand from the primary-copy site.

Most work focus on update propagation can be further classified as eager replication or lazy replication [24]. Eager replication requires that all replicas be updated before updating transaction commits while lazy replication sends updates to other replicas after the update transaction commits. The implementation of eager replication often requires distributed locking and two-phase commit. As a result, when the number of sites increases, the performance decreases drastically [24].

Recent efforts in [3, 9, 37, 38] place emphasis on lazy replication as it offers better performance at the expense of weaker consistency than 1SR. However, data inconsistency can occur when two transactions originating at different sites update their local copies and commit before propagating them to other sites. Hence, reconcilation is needed to resolve update differences that may be too costly to implement. To avoid a reconcilation, each data item is assigned a *primary copy* site and only transactions originating at the primary copy site are allowed to update [15]. Other sites have only read accesses. The distribution of data replicas across sites is represented in a *data placement graph* whose edges are directed from a primary site of a data item to a secondary site of the same data item. The serializability is guaranteed only if the graph is acyclic.

A consistency protocol must also ensure that the effects of site or communication failure and recovery are transparent to all replicas [6]. In case of a site failure, updates may be delayed. The *read-one-write-all-available* approach [6] allows writes to proceed to all available copies while keeping track of which site had missed a write. When the site is recovered, it can request up-to-date copies from any available site. In case of communication failure and the network is partitioned, the Quorum Consensus approaches

[6] allow writes to perform at majority sites of the network while reads are executed only at the local site.

2.3 Multidatabases

A multidatabase system (MDBS) integrates existing and possibly heterogeneous databases while preserving their local autonomy [44]. The motivation is to join individual and independently developed databases without any modification and hence to save the existing investment. The MDBS will act as a front end to local databases and facilitate any operation performing across them.

Local autonomy in an MDBS allows local databases to maintain complete control over their local resources. Each local database chooses to join or depart the MDBS at any time without major changes. They simply add global functions to access the system while their local functions remain unchanged. In addition, each local node determines what information is to be shared globally.

Due to local autonomy, each database joining an MDBS may be heterogeneous. They often have different data models and different query languages. The MDBS must be able to map various data models to a common and canonical model. In addition, semantically similar data may have different names and representations while different data may have identical names. The MDBS is responsible for resolving these differences and determining the semantics of data. This overhead may degrade the global performance significantly as it has no control over local databases.

There are two general approaches to resolve multidatabase heterogeneity [12]. In the first approach, local schemas are integrated to form a global schema representing

common data semantics. The process of integration is rather complicated, labor intensive and probably requires manual manipulation. The global schema is usually maintained at every site to allow simple and fast accesses to the data. However, the consistency problem may arise when there are updates at local databases [10].

In the second approach, the integration of local schemas is achieved through a common multidatabase language which interprets and transforms a query to data represented and maintained at local databases. There is no global view of shared data. Thus, a user must know the location and the representation of data being queried. The approach is naturally less transparent than the global schema approach, but it is more efficient and less complicated.

2.3.1 Summary Schemas Model

Summary Schemas Model (SSM) is proposed to resolve name differences among similar data in multidatabase systems [11]. It is an adjunct to multidatabase language systems for identifying semantically similar data that have different names and representations at various local databases. Basically, it uses word relationships defined in a thesaurus such as Roget's Thesaurus to build a hierarchical metadata of local access terms exported from underlying local databases. Users can submit imprecise queries at any site without knowing the location of access terms of the requested data. The SSM will map imprecise user terms with precise access terms found at some local databases.

Access terms can be related to each other as synonyms, hypernyms or hyponyms. Two terms having the same meaning are synonyms. The hypernym of a term has more

general meaning, but the hyponym of a term has more specific meaning. Hence a hypernym can have many hyponyms whereas a hyponym has only one hypernym. The SSM uses term relationships to form a hierarchical structure linking hypernyms and hyponyms while synonyms are linked on the same level.

The hierarchical structure of the SSM consists of local database schemas as leaf nodes and summary schemas formed as internal nodes. The architecture is shown in Figure 2.3. A schema at each node is a list of access terms. A summary schema node is created by mapping lower access terms to their corresponding hypernyms. Obviously, the summary schemas are smaller and more abstract than their lower schemas. Semantic similarity between any two terms are measured in terms of the *Semantic Distance Metric* (SDM) which is proportional to the number of hypernyms-hyponyms and synonyms linked between them. The SDM indicates how similar the terms are [11]. In other words, the smaller the SDM, the more similarity between two terms. When a user submits a query with imprecise terms at any node, the SSM will search upward or downward in the hierarchy for more precise term using the SDM. If the query is not originated at leaf nodes, the search is conducted downward to its hyponyms until it reaches a leaf node. If the query term does not match with its precise term at leaf nodes, the search is then upward to its parent node which contains its hypernym. The search is complete when the query terms are matched with precise terms at leaf nodes or at the root and no matching hypernym is found.

Figure 2.4 shows an example of SSM hierarchy for access terms related to personal income. Those access terms are exported from various local databases.

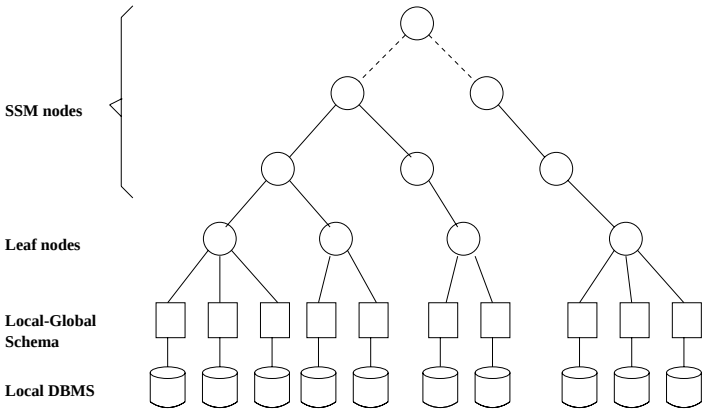

Fig. 2.3. SSM Architecture

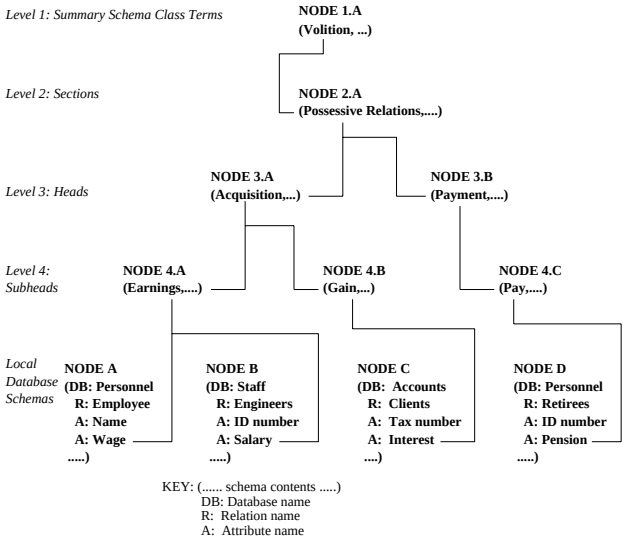

Fig. 2.4. An Example of SSM Hierarchy [11]

2.4 Security Policy

A security policy is a set of rules regulating and limiting the accesses to a set of objects by a set of subjects. In a typical computer system, the policy is usually defined and enforced by the operating system. It can be expressed through an access matrix [25] where rows represent subjects and columns represent objects. An entry at row S_i and column O_j contains a set of access rights that the subject S_i has to the object O_j. An empty entry at row S_i and column O_j indicates that the subject S_i cannot access the object O_j. Figure 2.5 shows an example of an access matrix where the letters `R`, `W`, and `X` in its entries represent `Read`, `Write` and `Execute` rights, respectively.

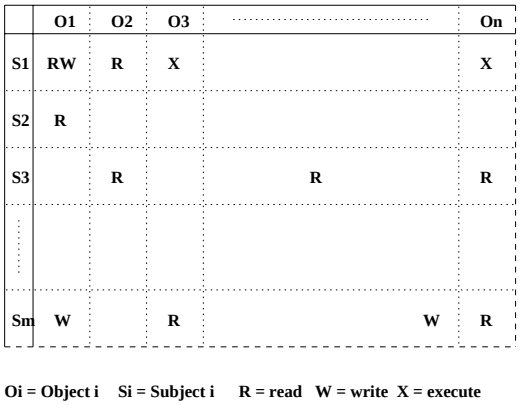

Fig. 2.5. Access Control Matrix

In most security policies, an object can be accessed only by a specific set of subjects, and a subject can access only a certain set of objects. As a result, most access matrices are sparse and keeping a lot of empty entries is a waste of space. Practically, a

security policy is expressed in a simpler and more efficient structure either as an access control list (ACL) or as a capability list (CPL) [47]. An ACL stores a matrix by columns and each object is associated with a list of subjects and their access rights used to access the object. A CPL stores a matrix by rows and each subject is associated with a list of objects and access rights that the subject can use to access each object in the list. An example is given as follows.

Assume that we have 3 subjects: S_1, S_2, S_3, and 4 objects: O_1, O_2, O_3, O_4. Also, let the letters `R`, `W`, and `X` stand for `Read`, `Write` and `Execute` rights, respectively. The ACL of this set of subjects and objects is:

- O_1 : (S_1, RWX),(S_2, R), (S_3, X)
- O_2 : (S_2, RWX)
- O_3 : (S_1, RWX),(S_2, RW)
- O_4 : (S_3, RWX),(S_1, X)

A corresponding CPL of the above ACL for the same set of subjects and objects is:

- S_1 : (O_1, RWX),(O_3, RWX), (O_4, X)
- S_2 : (O_2, RWX),(O_1, R), (O_3, RW)
- S_3 : (O_4, RWX),(O_1, X)

Two commonly used access control policies are discretionary access controls (DAC) and mandatory access controls (MAC) [14]. The main characteristics of DAC are the

ownership of an object and the capability of right delegation by an authorized subject who is allowed to delegate that right to another subject. In other words, DAC policy is owner-based administration of access rights. DAC policies are mostly represented as ACLs or capabilities.

On the contrary, MAC policy is a group-based policy where data is classified into groups and users must have proper privilege levels to access those data. It is also defined as a flow-control policy because it prevents information flow to objects of a lower classification. Examples of MAC policy are multilevel security and type enforcement policy and they are described below.

2.4.1 Multilevel Security

Multilevel security (MLS) policy [16] is defined in terms of a lattice of levels partially ordered according to the *dominance* ($\geq$) relation. If two security levels H and A satisfy $H \geq A$, then we say that level H dominates level A as shown in Figure 2.6. The figure also shows that both levels A and B dominate level L, but levels A and B are incomparable.

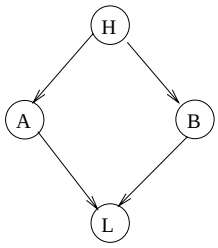

Fig. 2.6. Lattice of Security Levels

In MLS, each subject is assigned a level representing its *classification* and each object is assigned a level representing the *sensitivity* of the information it contains. MLS policy allows the information to flow only in an upward direction. That is, the information can flow from an entity (a subject or an object) of level X to another entity of level Y only if level Y dominates level X.

The Bell and LaPadula model [5] defines two properties. The first property, called *Simple Security Property*, allows a subject to read an object only if the security level of the subject dominates that of the object. The second property, called **-property*, allows a subject to write to an object only if the security level of the object dominates that of the subject.

In a typical computer system, overt information flow among entities can be prevented via authentication and access control mechanisms. For information flow policies such as MLS, in addition to direct communication between subjects via reads, writes or other permitted means, communication through other unintended means that violate the policy must also be controlled. Such a communication method is known as a *covert channel*. Figure 2.7 depicts both overt and covert channels.

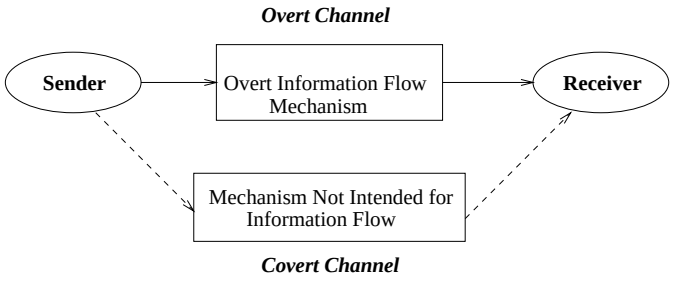

Fig. 2.7. Overt and Covert Channels

Covert channels are communication paths that use indirect means for transfering information among entities [2]. Hence, they can be misused to leak information to an unauthorized individual. Covert channels can arise through resources shared among multiple security levels when some activities of high-level users interfere with those of low-level users. Consider a scenario when there is only two users: a high-level user HIGH and a low-level user LOW using the system. Suppose a low-level file consists of only one block where HIGH can only read, but LOW can read and write to that block. Assume that reading or writing to a file requires an exclusive lock. Initially, HIGH acquires the lock and reads data from the block. At about the same time, LOW wants to write to the block, but cannot acquire the lock since HIGH is still holding it. Thus, LOW can infer that HIGH is reading the block. We say that the activity of HIGH interferes with that of LOW.

A system is guaranteed to be covert channel free if it satisfies the noninterference property [21]. The noninterference property requires that the view of the system through the outputs available to a subject cannot be affected by actions taken by subjects not dominated by the observing subject.

2.4.2 Type Enforcement Policy

A Type Enforcement policy [8] can be specified as a matrix where each row represents a domain in which a group of subjects may execute and each column represents a type of object accessible only by a particular domain. From the example shown in Figure 2.8, the letters `R, W, X` represent `Read, Write, Execute` rights, respectively. An non-empty entry at row D_i and column T_j containing at least one letter indicates

that every subject executing in domain D_i has corresponding access rights to all objects of type T_j. Even though some entries may be empty, the matrix is not large and sparse since the number of domains and types in the policy is much less than the number of individual subjects and objects.

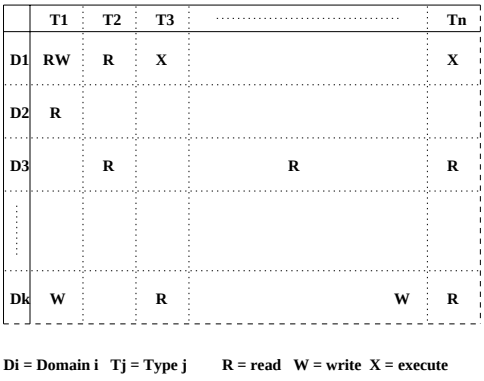

Fig. 2.8. Domain-Type Matrix

The policy is used for controlling the flow of information between groups of objects, not for controlling access to a particular object. Typically, it is used to separate a group of sensitive objects from the rest of the system, and allows only subjects of a particular domain, called *trusted domain*, to access those objects. Moreover, the policy can be used in conjunction with MLS by defining a specific set of subjects running in a trusted domain. Those subjects are granted some additional access rights beyond those allowed by MLS. However, the subjects must be *verified* to prevent any misuse of a given privilege. Other groups of subjects are unprivileged and can operate only at their assigned security level.

2.5 Conclusion

In this chapter, we presented general background on topics related to our research. First, we explained the basic concept of transaction processing and its essential mechanisms such as concurrency control, consistency and recovery. Next, we gave an overview of consistency control protocols for replicated databases as they are fundamental to our research in maintaining consistency of security policies in a distributed environment. Later, we described in detail a multidatabase system and its defining characteristics, local autonomy and heterogeneity. Moreover, we introduced the Summary Schemas Model since security requirements will be imposed in the model and the performance will be evaluated. Finally, we described multilevel security and type enforcement policies since their unique requirements have significant impact on the design and the performance of a database system running under them.

Chapter 3

Multilevel Log Manager

A database management system (DBMS) organizes and manages a collection of related data. A logical unit of database processing is represented as a transaction which is comprised of several operations and must be executed in its entirety or not at all. The concept of transaction provides a basic mechanism to ensure the consistency of the database whenever there is a change in the system.

The client-server architecture of a transaction processing system described in [23] consists of transaction managers (TMs), resource managers (RMs) and a log manager (LGM). Resource managers execute queries submitted from clients. Each query accesses some data items in the database. The transaction manager coordinates commitment and recovery of transactions. During recovery, resource managers may have to undo uncommitted transactions or redo committed transactions using log records describing all activities of RMs and TMs. A log manager maintains log records written by RMs and TMs.

In the design of a database system, any access to data items in the database must be authorized. Multilevel security (MLS) adds an extra authorization control since it requires that every data is given a security level and can be accessed solely by users or subjects with equal or higher security level. This special requirement of MLS

has influenced design choices in every component of a transaction processing system including a log manager.

In this chapter, we present our design and implementation of a multilevel log manager. Our log manager is part of the Secure TransActional Resources Database System (STAR-DBS) which is an experimental multilevel secure database system running under a secure operating system called Distributed Trusted Operating System (DTOS) [30]. DTOS supports multiple security policies including MLS and Type Enforcement. It also provides MLS enforcement for all components of the STAR-DBS [4, 39, 46, 50].

The remainder of this chapter is organized as follows. First, we describe related background such as log management, security issues in log management, DTOS and STAR-DBS systems. Second, we present the design and implementation of our multilevel log manager. We also explain experiments and discuss the results to evaluate the performance of our log manager. Third, we describe work related to the design of multilevel log managers. Finally, we conclude.

3.1 Background

In this section, we give background related to the design of multilevel log managers. First, we introduce the basic concept of log management and performance issues of a log manager. Next, we briefly explain the main features of DTOS and the implementation of STAR-DBS running on DTOS. Finally, we describe security issues related to log management.

3.1.1 Log Management

DEFINITION 3.1. *A log is a sequence of log records containing values of data before and after being modified by transactions.*

Logs are used by a recovery manager for undoing or redoing transactions during recovery from a failure. A log is typically implemented as a sequential file. Each log record is identified by a log sequence number (LSN) which tells the position of the individual log record in a log file. Since the log may be kept in several files, LSN is generally composed of the log file number and the byte offset from the beginning of the log file. This LSN is specifically called a physical LSN. A new log record is always appended to the end of the log. In addition, log files are often duplexed such that damage to the log caused by a single disk failure can be recovered from. Figure 3.1 illustrates how logs play a role when a data item is modified.

Writing log records directly to a log file for every update transaction will degrade the performance of the system significantly. Thus, in general, log records are buffered in volatile memory page. However, if the system fails while log records are in the buffer, those records will be lost. To ensure that no log record is lost, write-ahead log (WAL) protocol was implemented [32]. The protocol requires that a log record be present on disk before a memory page containing modification described in the log record is written to disk. It also requires that all log records generated by one transaction be on disk before the transaction is allowed to commit.

A log manager provides two principal functions: *Log_insert* and *Log_flush* [23]. *Log_insert* is used when a new log record is appended to the end of the log. The *Log_insert*

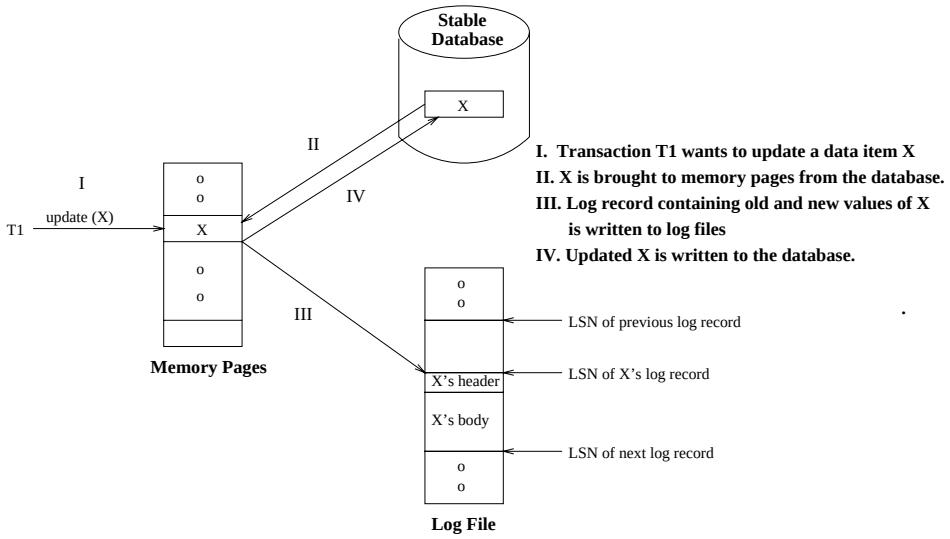

Fig. 3.1. Log Management

function has a log record and its record size as inputs and an LSN of the log record as an output. *Log_flush* is used to force all log records preceding and including the specified LSN to be written safely on disk. However, the function returns the maximum LSN of the log record that had been written on disk if it is greater than the specified LSN. The WAL protocol calls *Log_flush* to make sure that all log records created by a transaction are flushed to the log before the transaction commits. Interfaces of the two functions are shown below.

```
LSN log_insert(log_rec_t log_record, int log_record_size);

LSN log_flush (LSN specified_lsn);
```

3.1.2 The Performance of a Log Manager

The performance of a log manager has a significant impact to the performance of a database system since every update transaction generates log records to be appended to the log. Two main performance metrics for a log manager are log bandwidth and flush latency. Log bandwidth is the rate at which log records are appended to the log file by a log manager and can be measured in terms of the number of bytes per second. It indicates the throughput of a log manager. Hence, log bandwidth is an indication of how fast an update to a database can perform.

Flush latency is the time a log manager takes to complete a log flush request. Thus, the sooner a log record resides on disks, the shorter the flush latency. It indicates the response time of a log manager. Since a log flush request is usually made before an update transaction commits, low flush latency will lead to fast response time of the update transaction.

To reduce the flush latency, disk-write latency which mainly consists of the rotational delay and the seek time should be minimized. Write-Ahead Data Set (WADS) technique [20, 23] can be used to reduce disk-write latency. The technique reserves a cylinder on a pair of dedicated disks. When a new log record is created, it is written at the next sector of the current track coming under the disk head. Successive log records are written to different tracks of the WADS cylinder. After all tracks have been written, log records on those tracks are copied to the current end of the log at one time and all tracks are now free to accept a new log record. The WADS technique greatly improves

disk-write latency since the seek time is zero and the average rotational delay is about half of a sector. As a result, the flush latency is significantly reduced.

3.1.3 Distributed Trusted Operating System (DTOS)

DTOS [30] is an experimental prototype secure operating system developed at Secure Computing Corporation. DTOS prototype provides a security architecture that separates the definition of a security policy from the mechanism that enforces it so that the security policy can be changed without modifying the enforcement mechanism. The architecture has two parts. They are a Security Server and an object manager as shown in Figure 3.2.

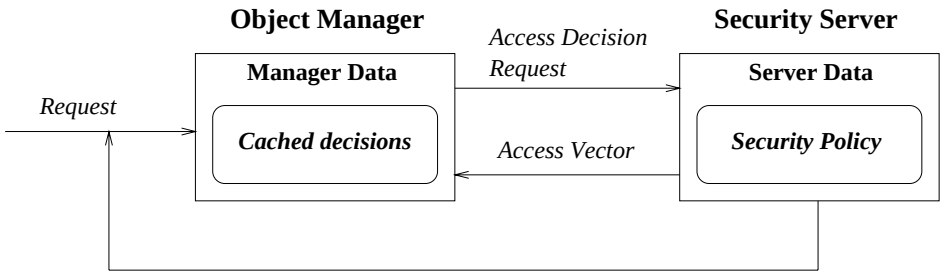

Fig. 3.2. DTOS Architecture

The Security Server defines a security policy consisting of access decisions. An object manager enforces the policy and controls all accesses to a collection of objects that it manages. When a subject wants to access an object managed by the object manager, the object manager sends access decision requests to the Security Server. In response,

the Security Server sends an access vector containing a set of access decisions for the subject when accessing the object. The object manager may cache the access vector for future use and, thus, avoids future interactions with the Security Server.

In implementation, DTOS has the modified Mach microkernel [1, 22] as the object manager. All Mach microkernel services have been modified to request access decisions from the Security Server before granting an access. DTOS implements a user-level Security Server so that a new security policy may be defined if necessary.

The DTOS prototype supports two security policies: multilevel security (MLS) and Type Enforcement policy. All subjects are regulated by both policies. To be independent of any policy, the object manager cannot contain any policy specific information. Thus, every object and subject in DTOS is labeled with a security identifier (SID). The Security Server defines a one-to-one mapping between an SID and a *security context* representing a security level in MLS, a type or domain in Type Enforcement policy and an owner of the object. The Security Server then defines the policy by determining access permissions between a pair of object SIDs and subject SIDs based on their security contexts kept at the Security Server. The object manager enforces the policy with the help of the Security Server. To reduce communication between the Security Server and the object manager, access permissions are cached at the object manager.

Object types implemented in DTOS include *tasks* and *threads*, and *ports*. Interprocess communication (IPC) among tasks and threads is supported through ports and messages. Thus, they are assigned proper SIDs so that communication among them is also controlled by permissions requested from the Security Server. In addition, DTOS file

systems are attached with appropriate security contexts and security policy is enforced for all operations on files using the Security Enhanced Lites Server.

3.1.4 STAR-DBS Prototype

STAR-DBS prototype is an experimental multilevel secure database system running on DTOS. DTOS features supporting the prototype include labeling objects and subjects according to multilevel security requirements. DTOS also provides a way to create a new domain for privileged subjects defined in STAR-DBS such as transaction managers and resource managers so that tasks and threads created by the manager are run in proper domains.

STAR-DBS prototype adopts a client-server architecture as shown in Figure 3.3. When a client executes a transaction program, a transaction starts by contacting a transaction manager (TM). The TM assigns a unique transaction identifier to the transaction. The transaction then makes relational query requests (such as read, insert, delete and update) to one or more resource managers (RMs) who execute the requests. When the transaction is complete, it contacts the TM again to make a request that its work be committed.

Each RM is unprivileged and provides services to transactions at its own security level only. It is associated with a unique resource manager identifier. An RM implements a set of relational operations and provides interfaces to access tuples. Usually, a portion of the database is brought from a disk to a page in memory to avoid a disk access for every read and write to data items. Before the RM accesses data items in a page, it makes a pin request to the level buffer manager (LBM) to ensure that the page is

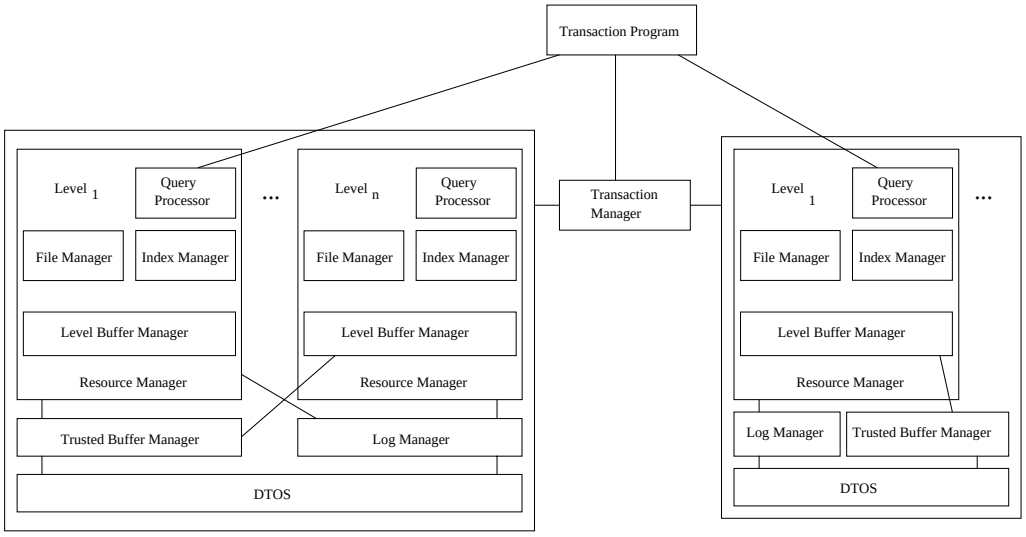

Fig. 3.3. Architecture of STAR-DBS Prototype

not replaced during its access. The LBM satisfies the request by coordinating with the trusted buffer manager (TBM). The TBM controls the movement of data between persistent and volatile portions of the database for all security levels.

The log manager (LGM) is a trusted component providing logging related functions to all security levels. It writes uninterpreted undo/redo records to the log on behalf of RMs, writes commit and abort records generated by the TM, and controls flushing of log records to disks. It also supports retrieval of log records during the recovery process.

3.1.5 Covert Channels in Multilevel Log Management

Regarding MLS, covert channels can be found where resources are shared among security levels. A trusted application running under MLS has to identify all possible covert channels that may exist. The application must also be designed to avoid or

prevent those channels. Three covert channels concerning the design of a multilevel log manager are identified in [39]. They are *insert channel*, *flush channel*, and *disk scheduling channel*. We describe them in turn as follows.

3.1.5.1 Insert Channel

If an LGM assigns a unique and monotonically increasing LSN for every new log record inserted regardless of the log record's security level, a log record insertion at one level can affect the LSN returned to another level. This is called *insert channel*. For example, initially, a low-level client inserts a log record and receives an LSN from the log manager. Then, a high-level client is scheduled to run. It either inserts a log record (sending bit 1) and gets the next assigned LSN, or it does nothing (sending bit 0) and causes no change in the LSN. Next, the low-level client inserts another log record and notices the LSN it receives. This action of the high-level client violates the noninterference property since the high-level client interferes with the outputs seen by the low-level client.

3.1.5.2 Flush Channel

A *flush channel* exists if a log flush request from a high-level client results in flushing log records from a low-level client as well. As a result, the time taken to finish a log flush request for the low-level client is different when there is an intervening log flush request from the high-level client (sending bit 1) and when there is no request from the high-level client (sending bit 0). Consider two situations. First, both the high-level client and the low-level client issue log flush requests to the log manager. Second,

only the low-level client issues a log flush request. The flush latency measured by the low-level client in the first situation would be less than the one measured in the second situation because, in the first situation, log records of the low-level client have already been flushed to disk when the high-level client flushes its log records. Whereas, in the second situation, the low-level client has to wait for its log records to be written to disk.

3.1.5.3 Disk Scheduling Channel

When a disk is shared across multiple security levels, a *disk scheduling channel* [27, 51] may exist mainly because of optimization techniques used by most disk arm scheduling to reduce the average seek time. The most commonly used disk arm scheduling is the *elevator* algorithm because it gives both efficiency and fairness. In this algorithm, any pending seek request receives a service and is complete when the disk arm is moved over it. As a result, a completion order of disk seek requests of one level can be interfered by the issuing order of another level. This would also result in the timing difference of the same set of seek requests if they are issued twice.

An example given in [39] is briefly described here. Consider a disk partition containing contiguous cylinders shared for reading by two log files of different security levels : HIGH and LOW. Assume that the contiguous cylinders are ranged from 11 through 50. Initially, HIGH issues a seek request to either cylinder 11 (sending bit 0) or cylinder 50 (sending bit 1), and waits for the seek to complete. LOW then issues a seek request to cylinder 23 and 47. If HIGH had issued its seek request to cylinder 11, the LOW completion order would be 23 and 47. But, if HIGH had issued its seek request to cylinder 50, the LOW completion order is reversed. Hence, LOW would see the time

difference when measuring the time it takes for a seek to complete. This covert channel exists in log management when log records are written to the same disk partition shared by both LOW and HIGH since the time taken to respond to a log flush request by LOW would be different if HIGH also issues its log flush request although LOW and HIGH have separate buffers for their log records.

3.2 Design

The main objectives in designing a multilevel log manager for the STAR-DBS prototype are to prevent or avoid the three covert channels previously described, and to achieve good performance. The basic idea of the design is to avoid having log records of multiple security levels shared common resources. Specifically, we require that each security level has its own set of buffers called Level Buffer Chain (LBC), and has its own log files called Level Log Files (LLF). The design consists of three steps as follows.

I. When a new log record is created, it is placed in a LBC buffer of the same security level of that log record.

II. When a LBC buffer containing log records is full, the buffer is written to LLF files of the same security level as the level of the buffer.

III. Log records in a partially filled LBC buffer of each security level are continuously written to a separate block on WADS disks in a round-robin fashion among all levels currently logged in.

The diagram in Figure 3.4 shows steps I, II and III, respectively.

With separate buffers and log files, when log records in a LBC buffer of one level are written to a LLF file of the same level, it does not cause log records in a LBC buffer of another level to be written as well. Hence, there is no *flush channel*. In addition, since writing a LBC buffer to a LLF file is executed separately for each security level, an independent physical LSN can be assigned for each level in the same way that it is assigned by a typical log manager. Thus, *insert channel* does not exist since there is no interference across security levels through the assignment of physical LSNs.

In this design, a *disk scheduling channel* exists if log files of multiple security levels are located on the same disk partition. To prevent this channel, either a secure multilevel file system is implemented or log files of each security level must be located on different disk partition.

The purpose of writing log records to WADS disks in the step III is to minimize flush latency. However, the WADS technique described in [23] requires that log records be copied from WADS disks to log files after all blocks on disks have been used up, and hence no log record can be written to WADS disks until the copy is done. We propose that flush latency can be minimized by writing log records in LBC buffers to both WADS disks and log files simultaneously. To effectively reduce flush latency, we require that writing log records to WADS disks must be fast and be performed continuously regardless of the number of log records in a LBC buffer. Consequently, log records are likely to be on WADS disks prior to log files, and the acknowledgement to a log flush request would be faster than that of the same log flush request when log records are written to log files only. Since WADS disks are shared among multiple security levels, it is possible to have

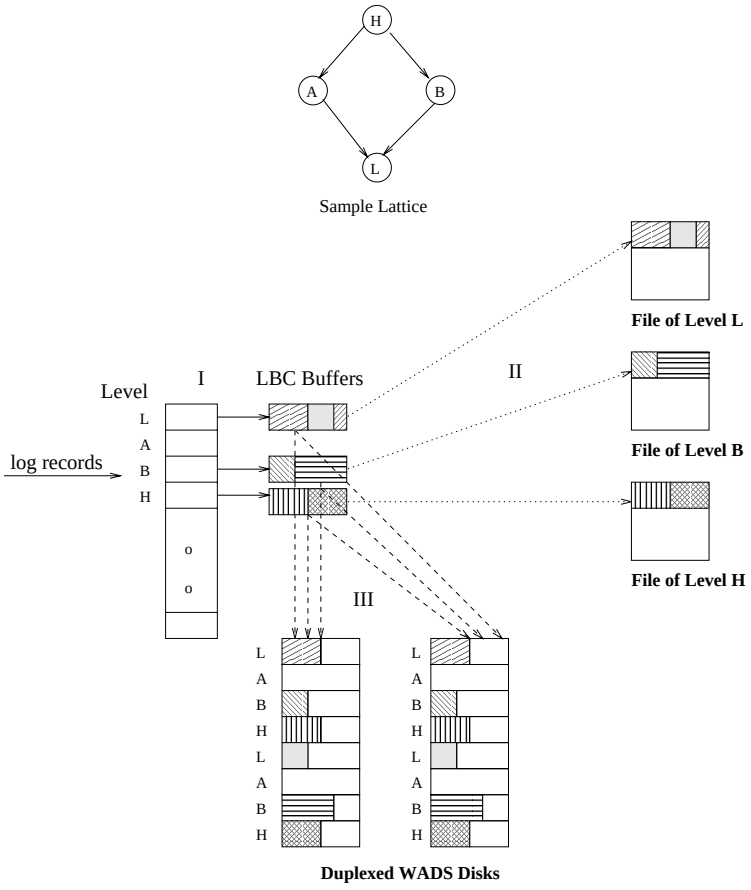

Fig. 3.4. Design of Proposed Multilevel Log Manager

a flush channel. To prevent this channel, writing log records to WADS disks must be scheduled in a round-robin fashion among all active security levels. In addition, writing log records to WADS disks is necessary only when log records in LBC buffers have not yet been written to log files. Thus, blocks on WADS disks can be reused.

In the diagram of Figure 3.4, all four security levels: H, A, B and L, are active. The figure shows that log records are inserted into LBC buffers of Level H, Level B and Level L, whereas there is no log record in LBC buffers of Level A. As can be seen that LBC buffers of each security level are written to duplexed WADS disks in a round-robin fashion even though they are empty or partially filled.

3.3 Implementation

In the implementation of our multilevel log manager[1], there are some important issues that we must consider. One issue is enforcing multilevel security in the log manager using DTOS services. Another issue is to avoid or prevent a covert channel that may occur between the time log records are inserted by clients until they are persistently stored on disk. The final issue is to achieve good performance (high log bandwidth and low flush latency) of the design in spite of extra time and space overhead. Each issue is discussed in turn below.

3.3.1 DTOS Features used for Log Management

DTOS features are crucial for the implementation of our multilevel log manager since the log manager is required to be in a trusted domain so that it can serve clients

[1]Part of our implementation is based on the code of the log manager proposed in [39].

from all security levels. The log manager is also required to have exclusive access to raw disk partitions for the implementation of individual LLF files and WADS disks.

DTOS features are used to create a *log manager* domain, and to label raw disk partitions of LLF files and WADS disks. They also give permissions to the log manager for communicating with all other domains in the STAR-DBS system. Such domains are the *transaction manager* domain, the *resource manager* domain and the *trusted buffer manager* domain. These domains communicate with the log manager via its advertised ports. The log manager allocates and advertises a port when a new level is created so that all requests of this level are sent to the log manager through this port. To enforce multilevel security, the log manager uses DTOS features to verify the security level of clients requesting the service.

3.3.2 Writing Log Records to Log Files

The rate of log record insertion to LBC buffers of each security level may be varied. Hence, two different security levels might not have their full LBC buffers written to their LLF files at the same time. A timing channel is likely if there is only one thread responsible for writing full LBC buffers for every security level as one level has to wait for another level to finish. As a result, each security level has its own thread for writing its LBC buffers to LLF files.

An LLF file for each security level is implemented as a disk-based log. Specifically, a raw disk partition is exclusively allocated for an LLF file, and LBC buffers are written directly to the partition. The write is also synchronous to ensure that the write is complete only after log records are on disk.

If LLF files of all security levels share the same disk, *disk scheduling channel* is present. In the implementation of our log manager, LLF files of each security level are located at different disk partitions and, if possible, also on different disk drives. Unless writing to disk can be carried out in a secure fashion by the underlying operating system, the number of active security levels in the implementation is limited to the number of disk drives the system has.

3.3.3 WADS Implementation

The main idea of the WADS technique is to write log records to disk as fast as possible to minimize flush latency. The technique described in [23] reduces disk-write latency by having minimum seek time and rotational delay since log records can be written to any sector of the next available track on dedicated cylinders of a disk. However, using modern disk drives such as Small Computer System Interface (SCSI) drives in our log manager, disk-write latency is minimized by adhering to the following requirements and employing special writing routines.

First, we require that two dedicated disk drives are exclusively used as the duplexed WADS disks although only a few blocks are actually needed. This ensures that the only disk accesses to each WADS disk are write requests from the log manager. Next, a raw disk partition is allocated on each WADS disk in order to have contiguous blocks in the partition. Hence, doing successive writes to the partition sequentially would give possible minimum disk write latency. This partition is called the WADS partition. Finally, Mach asynchronous write routines called *device_write_request()* and *device_write_reply()* are used for writing log records to duplexed WADS partitions. The *device_write_request()*

routine sends a write request to the WADS disk and the upcall *device_write_reply()* is called when the write is complete. Using this method is much faster than using the redirected I/O commands through the Lites Server because a write request is sent directly to the kernel bypassing a lot of system call overhead. Furthermore, since they are asynchronous, multiple write requests can be streamed to the WADS disk without using multiple threads required for synchronous writes. Thus, it requires only one thread to carry out all writes and no synchronization is needed. The tag queue feature of most current SCSI disks also allows several write requests waiting within the WADS disk so that writing to the WADS disk can be performed continuously. Thus, it helps reduce disk-write latency significantly.

3.3.4 Writing Log Records to WADS Disks

First, for each active security level, log records in LBC buffers are copied to *two* allocated buffers called WADS buffers. Second, the two WADS buffers are written independently and simultaneously to duplexed WADS partitions using Mach asynchronous write routines mentioned above. There are two writer threads. Each individual thread is responsible for writing one buffer to one WADS disk. Two buffers are used because, if one buffer is used, writing the buffer to two disks cannot be done at the same time[2], and hence writing the buffer to one disk can start only after the buffer has been written to the other disk.

[2]This is known as buffer sharing where a buffer is locked by a disk process until the writing is done.

To avoid a *flush channel* due to writing log records to WADS disks, the WADS buffers must be written continuously even if they are empty. Since the number of blocks on WADS partitions is limited, eventually these blocks will be overwritten. The log manager must ensure that log records on a WADS block are on LLF files before overwriting the block. Hence, the log manager keeps track of the status of LBC buffers and WADS blocks.

3.4 Experiments

A set of experiments is conducted to measure the performance of our log manager implementation. Two performance metrics are log bandwidth and flush latency as they represent the throughput and the response time of the log manager. To have a baseline for comparing the performance, we implement a multilevel baseline log manager which is functionally analogous to having an individual log manager for each security level.

3.4.1 Implementation of A Multilevel Baseline Log Manager

The implementation of a multilevel baseline log manager can be derived from the implementation of our log manager by implementing only the first two steps. Specifically, new log records of one security level are placed in its own LBC buffers before being written to its LLF files only when the buffers are full. No WADS disk are used. Each security level also has its own thread to write its full LBC buffer to individual LLF files located on its own disk partition.

The three covert channels do not exist in this implementation. There is no insert channel as physical LSNs of each security level can be independently assigned. Flush

channel does not exist because a log flush request results in flushing log records of its own security level only. The disk scheduling channel is solved since each security level has its own partition for its LLF files.

3.4.2 Experiment Environment

The experiments are done on a PC with 200 MHz Pentium processor and 64 MB main memory. The machine is run under DTOS as its security features are used for the implementation of both log managers. The disk system consists of four Ultra Wide SCSI Western Digital WDE4550 drives connected to a SCSI Adaptec 2940UW controller. Each drive has its cache turned off so that disk-write latency measured is the time actually taken to write data to the media, not the time to place data in the disk cache.

Among four disk drives we have, two drives are exclusively allocated for duplexed WADS disks to make sure that there is no other disk access request to WADS disks except from the log manager. Each of the other two drives has two partitions where one partition is allocated for LLF files of one security level. In other words, each security level has its own disk partition. Hence, we can have four active security levels in our experiments.

3.4.3 Workload and Measurement

In our experiments, we measure two performance metrics: log bandwidth and flush latency. Log bandwidth measures the rate of log record insertion processed by a

log manager in terms of the number of bytes per second. Flush latency measures the response time of a log flush request in milliseconds.

For all measurements, the sizes of both LBC buffers and WADS buffers are 32 Kbytes since it gives the highest bandwidth when writing to the underlying disk configuration. In each run of the experiments, the workload is composed of logging activities of four clients; each having two threads. All four clients may log in at the same or different security level and they start running at almost the same time. In addition, each client contributes approximately equal workload regardless of its security level. To avoid delay due to network communication, all clients are running on the same machine as the log manager.

We conduct two experiments. In the first experiment, we measure log bandwidth and flush latency of both the baseline log manager and our design for a single security level. In the second experiment, we measure log bandwidth and flush latency of our log manager when the number of security levels is 1, 2, and 4. In other words, when the number of security level is one, all four clients are at the same security level. Likewise, when the number of security levels is two, two clients log in at one security level and the other two clients log in at another level. For four security levels, each client logs in at different security level.

3.4.3.1 Log Bandwidth Measurement

In log bandwidth measurements, each thread of a client is run for the same number of repetitions. To create a stream of log records to the log manager, in each repetition, a thread generates and inserts a number of log records to the log manager. Then, it

waits for approximately 10 milliseconds to allow other threads to run. To simplify our workload description, we define two terms: *client insertion rate* and *offered insertion rate*. The client insertion rate is the rate of log record insertion generated by all threads of a client and it is computed using the following equation:

$$C = \frac{R * I * S}{T} \tag{3.1}$$

where

C is the client insertion rate,

R is the number of running threads of a client,

I is the number of log records inserted in one repetition,

S is the size of the log record inserted, and

T is the average measured waiting time of all threads of a client.

The offered insertion rate is the summation of the client insertion rate from all four clients. The log bandwidth is then measured at various offered insertion rate. From the equation above, the client insertion rate is affected by all four parameters. In our experiments, the number of running threads for a client is fixed at two, and the log record size is 180 bytes. For each client, the waiting time is measured and averaged between its two threads since each thread may be scheduled to wait shorter or longer than 10 milliseconds which is the time it is set to wait.

The only parameter that can be varied to generate different client insertion rates is the number of log record insertion in one repetition. Each client insertion rate is then combined to produce various offered insertion rates. In the experiments, the number of

log record insertions in one repetition starts from one and is incremented by one until the log manager reaches its maximum bandwidth.

3.4.3.2 Flush Latency Measurement

Workload for flush latency measurement is slightly different from workload for log bandwidth measurement. Two threads of a client generate client insertion rate as in log bandwidth measurement. However, at the end of a repetition, one of the two threads makes a log flush request to the log manager via the *Log_flush()* function. The call to the *Log_flush()* returns only when the latest inserted log record has been placed on its LLF files or on WADS partitions. There is only one *Log_flush()* call for each repetition, and flush latency is measured for every call.

Similar to log bandwidth measurement, flush latency is measured at various offered insertion rates by varying the number of log record insertions in one repetition, while other parameters are the same as in log bandwidth measurement. In summary, input workload is offered insertion rate and two outputs are log bandwidth and flush latency.

3.4.3.3 Reported Metrics

During each run of the experiments, *Log_insert()* and *Log_flush()* routines called by threads at a client involve interprocess communication (IPC) where the actual calls are carried out by the log manager. To factor out the effect of the IPC, a measured output is reported at the log manager rather than at the client. In addition, only the mean value of every data measured is reported.

In every run, there are two transient periods: one is at the beginning and the other is at the end. The transient period at the beginning occurs when the log manager starts and all LBC buffers are empty. This results in high log bandwidth in early measurement since log records are just copied into LBC buffers, but have not yet been written to LLF files or WADS disks. The transient period at the end is due to early exit by one or more threads of clients. Even though all threads start running at approximately the same time, one thread could finish log record insertion before another. At the end of a run, the offered insertion rate is gradually dropped and results in low log bandwidth. Hence, both transient periods must be eliminated before all measured data is averaged.

Generally, the average measured data of an experiment is reported when the system is in a steady state. In our experiments, the total number of inserted log records varies between 200,000 to 500,000 records per run. Transient periods at the beginning and at the end of each run are removed. Every measured data point reported is the mean value of several runs. In all cases, the 95% confidence interval for the measurement (assuming a normal distribution) is within $\pm 10\%$ of the sample mean value.

3.5 Results and Discussion

For each graph shown in this section, SIMPLE represents the multilevel baseline log manager and WADS represents our design.

3.5.1 Single Level Log Bandwidth

The measured log bandwidth for a single security level of both multilevel log managers is shown in Figure 3.5.

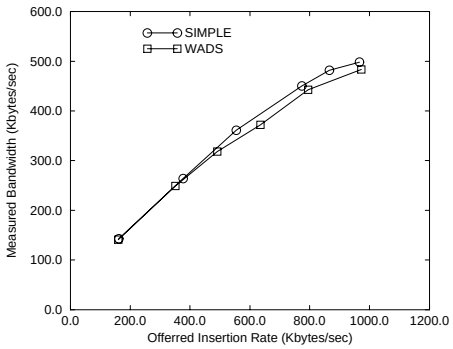

Fig. 3.5. Single Level Log Bandwidth

As can be seen, log bandwidth measured for both log managers are almost the same except when the offered insertion rate is high. Specifically, our designed log manager has slightly lower log bandwidth (within 5%) than that of the baseline log manager. This is because our log manager has more work to do as it has to write log records to two places (LLF files and WADS disks), while the baseline log manager writes them to only one place (LLF files). The measurement also illustrates that our log manager can process log record insertion almost as fast as the baseline log manager. The highest log bandwidth of both log managers measured is approximately 500 KBytes per second.

3.5.2 Single Level Log Flush Latency

Figure 3.6 shows the measured flush latency for a single security level of both log managers.

From the graph shown, the measured flush latency of the baseline log manager decreases inversely with the offered insertion rate. The reason for this is that, in the

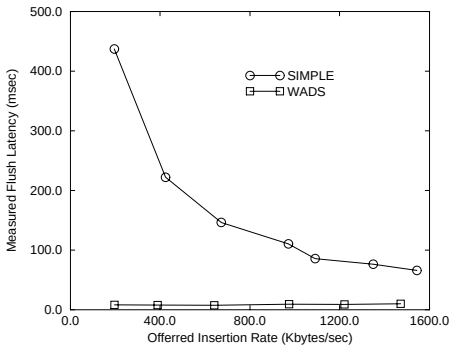

Fig. 3.6. Single Level Log Flush Latency

baseline log manager, a call to *log_flush()* has to wait for the LBC buffer to be full before it is written to the LLF files. Hence, if the offered insertion rate is higher, it takes less time to fill a buffer and complete a log flush call which results in lower flush latency. The lowest flush latency measured in this experiment is about 65.2 milliseconds.

On the other hand, the measured flush latency of our design is independent of the offered insertion rate since a call to *log flush* can return after log records are written to a WADS disk. The call does not have to wait for the LBC buffer to be full and be written to disk. As a result, the time taken to complete a log flush call depends mostly on how fast log records are written to WADS disks. Since writing to WADS disks is performed continuously regardless of the offered insertion rate, the flush latency is also constant with respect to the offered insertion rate. The average measured flush latency is about 8.3 milliseconds which is much lower than that of the baseline log manager.

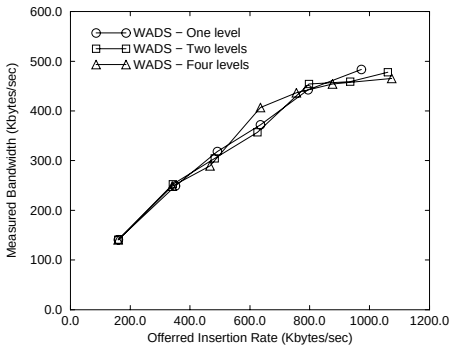

Fig. 3.7. Multilevel Log Bandwidth

3.5.3 Multilevel Log Bandwidth

The measured log bandwidth of our designed multilevel log manager for various numbers of active security levels is shown in Figure 3.7. The graph shows that measured log bandwidth is largely independent of the number of active security levels and depends only on the offered insertion rate. The highest log bandwidth measured is about the same as that of single-level case.

3.5.4 Multilevel Log Flush Latency

Figure 3.8 shows measured flush latency of our designed log manager for various number of active security levels.

As can be seen, measured flush latency is affected by the number of active security levels. In other words, when the number of security levels increases from 1 to 2, measured flush latency increases from 8.3 milliseconds to about 13.3 milliseconds. Moreover, when there are four active security levels, measured flush latency increases to 15.2 milliseconds.

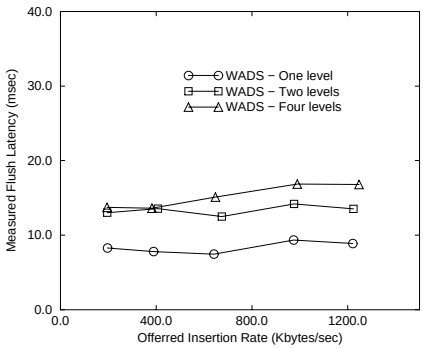

Fig. 3.8. Multilevel Log Flush Latency

This is because writing log records to the WADS disks is scheduled in a round-robin fashion among all present security levels. Thus, as there are more active security levels, each level has to wait longer for its turn to write its log records to WADS disks. In addition, when offered insertion rate is low, measured flush latency is almost the same for the case of two and four active security levels.

3.5.5 Discussion

The baseline multilevel log manager is used as the baseline for our performance measurement since it is a straightforward solution to multilevel log management. For single security level experiment, measured log bandwidth of the designed log manager indicates that additional overhead of the WADS technique has a minimal effect on throughput. On the other hand, the WADS technique helps improve measured flush latency of the designed log manager significantly. The results also verify that measured flush latency of the designed log manager does not depend on log record insertion rate.

For the multiple security level experiment, the results of the experiments show that the designed multilevel log manager supports multiple security levels since measured log bandwidth depends only on offered insertion rate generated by all clients of different levels, and it is not affected by the number of active security levels.

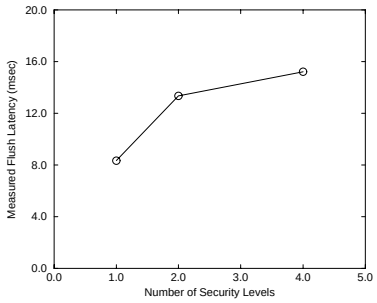

Fig. 3.9. Log Flush Latency with Varied Number of Security Levels

If measured flush latency depends only on the time of writing log records of each security level to WADS disks and each security level has its log records written in a round-robin fashion, we would expect that measured flush latency increases linearly with the number of active security levels. However, the increase is not linear since the results for four active security levels illustrates some degree of dependency on offerred insertion rate. Figure 3.9 shows that measured flush latency is slightly affected by the number of security levels. A reasonable explanation is that, when there are four active security levels, the flush latency of one security level is affected by the action of writing log records to LLF files by another security level since LLF files of two security levels

are sharing the same disk drive in our experiments. To eliminate this effect, LLF files of each security level must be located on different drives.

3.6 Related Work

The log manager proposed in [39] assumes that every security level is defined in a lattice. It then uses the dominance relationships among active security levels to create a set of security paths. Each node in a security path represents a security level. Each security path is constructed from nodes arranged from the lowest security level to the highest security level. Based on the lattice shown in Figure 3.10 below, two security paths constructed are $L \rightarrow A \rightarrow H$ and $L \rightarrow B \rightarrow H$.

The log manager of [39] operates in two phases as shown in Figure 3.10. The first phase is called the collection phase. In this phase, log records are collected from LBC buffers of low security level to those of high security level within a particular security path into WADS buffers. The reason for collecting log records this way is to avoid interference between security levels since each security level does not know when its log records are collected. As a result, a WADS buffer may contain log records of more than one security level. Also, it is likely that a WADS buffer may be full, partially filled or even empty. Finally, WADS buffers are continuously written to duplexed WADS partitions until all blocks are written.

The second phase is called the compaction phase. In this phase, each block in one of duplexed WADS partitions is read into a buffer. Then, log records that may be present in the buffer are separated according to its security level and they are written to its corresponding LLF files.

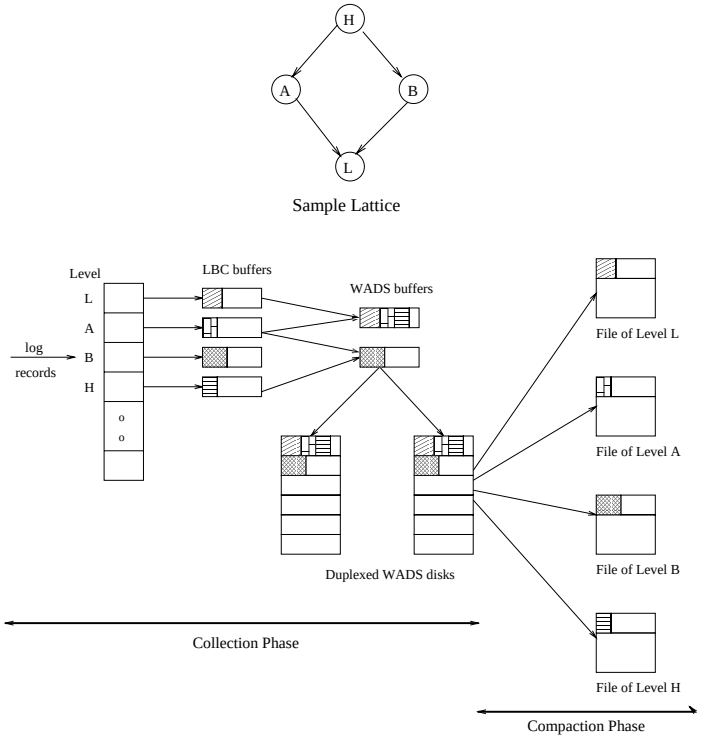

Fig. 3.10. Design of Multilevel Log Manager in [39]

The performance reported in [39] are log bandwidth of 39.06 Kbytes per second and flush latency of 2.46 milliseconds. This log bandwidth is lower than that of our designed log manager due to several factors including slow disks and disk controller (SCSI-2 Complaint) used in its experiments, redirected write command through the Lites Server and a required compaction phase. Upon investigation, we found that flush latency was incorrectly measured since a log flush request returns before a WADS buffer containing log records is written to duplexed WADS partitions.

In addition, our designed log manager gives higher log bandwidth than that reported by [39] mainly because our log manager does not have the compaction phase and writing to WADS partitions using Mach asynchronous write routines is faster.

3.7 Chapter Conclusion

This chapter presented a new design and implementation of a multilevel log manager for the STAR-DBS prototype. The proposed design is simpler and more efficient than the log manager proposed in [39] as illustrated in our experimental results. The results also show that, for single security level, log bandwidth of our log manager is comparable to that of the baseline log manager. But, flush latency of our log manager is lower and independent of log record insertion rate. For multiple security levels, log bandwidth of our log manager is not affected by the number of active security levels and its flush latency increases sublinearly with the number of active security levels. One additional benefit of our design is that physical LSNs are used in the same way as those in a typical log manager.

Security mechanisms provided by DTOS are very useful in the implementation of our designed log manager since they help isolate subjects of different security levels and provide secure communication among subjects. They also provide a trusted domain for the log manager so that the log manager can serve clients from all security levels.

Chapter 4

Security Policy Consistency

Accessing resources in a computer system is governed by access control rules defined by a security policy. The policy is typically defined and enforced by the operating system.

In the Distributed Trusted Operating System (DTOS) [30] and Flask [45] architecture, the definition of the policy and the mechanism that enforces it are separated into two components. The first component is the Security Server (SS) which defines the security policy and maintains it as a small database. The second component is the Object Manager (OM) which enforces the security policy by ensuring that every access to its resources is authorized by the Security Server. As a result, the policy can be changed by modifying the policy database at the SS, and hence there is no need to modify the OM so that it can enforce a new policy. To reduce access time, part of the policy may be cached at the OM. When there is a change in the policy at the SS, a replica at the OM must be invalidated. A new replica at the OM is created on demand from the primary copy at the SS. Figure 4.1 illustrates a snapshot of the interaction between the SS and an OM.

We consider a system consisting of the SS and multiple OMs located on different machines. We assume that the SS maintains the primary copy of the policy and its replica may be cached by an OM at each site. Any change in the policy is initiated only

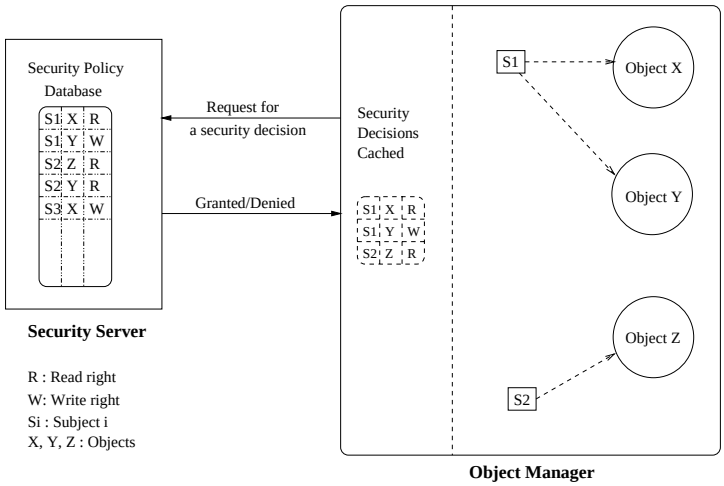

Fig. 4.1. Interaction between the SS and an OM

at the SS and OMs are not allowed to modify the policy. The system can be modeled as a replicated database. The model uses a centralized approach for enforcing security policy since changes in the policy must be restricted to one particular site and can be made only by specific sets of privileged users. If several sites are allowed to change the policy simultaneously, not only must we reconcile the order of changes, but we may also have to resolve possible conflicts.

All replicas of the policy must be consistent when the policy is changed so that accesses to objects or resources are not compromised [41]. Figure 4.2 shows an example of a possible security violation when a policy is changed at the SS, but its cached replica of the policy at the OM does not change accordingly. The figure illustrates that the SS has two policy decisions: P1 and P2. The two decisions contradict each other and so must not both hold at the same time. The decision P1 says that the object X is accessible to the subject S1, and the object Y is inaccessible to any subject. The decision P2 says

that the object X is inaccessible and the object Y is accessible. Suppose that the policy decision at the SS is initially P1. The OM then makes a request for X and keeps the decision in its cache. Later, the policy decision at the SS is changed to P2. As a result, the SS sends a message to the OM to flush the cache. However, before the OM gets the flush message, it makes a request for Y to the SS and caches it since the policy at the SS has changed. At this moment, the OM can access both X and Y which is never allowed in either P1 or P2.

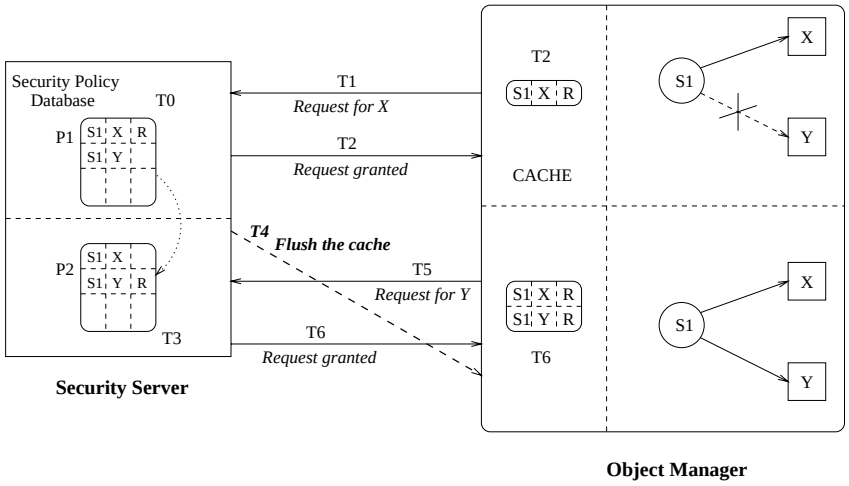

Fig. 4.2. An example of Security Violation

To ensure consistency among replicas of the security policy, strong consistency such as one-copy serializability (1SR) [6] is required since it guarantees one-copy view of the policy regardless of which copy is accessed.

Most work on maintaining consistency of replicas focus on update propagation protocols such as eager replication or lazy replication [3, 9, 15, 24]. However, we investigate invalidation-based protocols which require that all replicas at OMs be invalidated before the primary copy at the SS is updated. The motivation stems from several factors including:

- Invalidation-based as well as eager replication protocols guarantee one-copy serializability.

- The protocols offer higher performance than eager replication since it incurs less overhead for maintaining consistency. Eager replication incurs too much overhead at OMs due to required two-phase commit and deadlock detection mechanism. Hence, its access time increases significantly [24]. On the other hand, lazy replication mostly used for replicated databases has better performance than eager replication. But, it allows transient inconsistency among replicas which is undesirable for maintaining consistency of security policy databases.

- When updating replicated data in a typical database, aborted update transactions can be undone or redone, and data inconsistency among replicas is allowed for some period of time without compromising their security. However, when changing the security policy by update transactions at the SS is aborted, some OMs may retain the old copy of the policy while others may have received a new copy of the policy. Hence, the policy may be compromised at some OMs. Invalidation-based protocols invalidate the old copy of the policy during the update. A new policy is replicated

at OMs only after the update transaction at the SS commits. Thus, the protocols guarantee no inconsistency among the replicas at OMs.

- To recover from aborted transactions, both the SS and OMs may have to keep logs of policy changes at their sites. However, with invalidation-based protocols, no rollback is necessary at OMs, and thus OMs are not required to keep a log. Undo-redo operations for aborted update transactions can be done using logs kept only at the SS. Moreover, if an OM fails, no recovery at OM is required at all.

For security concerns, the requirement for strong consistency of the security policy is unquestionable, but the requirement for good performance is also desirable. The concept of invalidating replicas has been used in both DTOS and Flask architecture [30, 45]. In this chapter, we present two invalidation-based consistency protocols: Invalidation Lock-Based Consistency Protocol[1] (ILBP) and Invalidation Timestamp-Based Consistency Protocol (ITBP). The goal is to evaluate and compare the performance of the ILBP and the proposed ITBP protocol. We compare the performance of the two protocols with that of eager replication under various degrees of workload.

The remainder of this chapter is organized as follows. First, we explain the system model of the Security Server and multiple Object Managers. Next, we present two consistency protocols: Invalidation Lock-Based Consistency Protocol and the proposed Invalidation Timestamp-Based Consistency Protocol. Then, we describe eager replication which the performance of the two protocols is compared against. Each protocol is explained based on types of messages sent and received between the SS and OMs.

[1]The protocol was called Transactional Consistency Protocol in [43]

We also describe the simulation model and discuss the simulation results. Then, we explain work related to consistency protocols for maintaining consistency of security policy. Finally, we conclude the chapter.

4.1 Proposed System Model

The system is modeled as a replicated database. The model is composed of one security server (SS) and multiple object managers (OMs) as shown in Figure 4.3. The model does not rely on any specific security policy.

A security policy can be expressed as a set of (subject-id, object-id, permissions). A subject-id may represent a user, a group, a role or other entity as specifically defined by the policy. An object-id may represent a resource in the system. Permissions represent a set of access rights defined by the policy. Since each machine may assign identifiers of its own subjects and objects independently, we assume that there are identity mappings between subjects and objects among machines sharing the same identities of those subjects and objects. Those mappings can be specified by the policy.

The SS and OMs are communicating via messages. We assume that messages are sent via a secure and reliable communication so that they are not lost or arrive out of order. Thus, we can assume that messages arrive in the order that they are sent. In addition, OMs must always honor by invalidating or updating their replicas when they receive revocation or update messages from the SS, respectively. We define the body of a message as a tuple of (*subject-id, object-id, permissions, message-type, sender*) where a set of message types is specifically defined by each protocol.

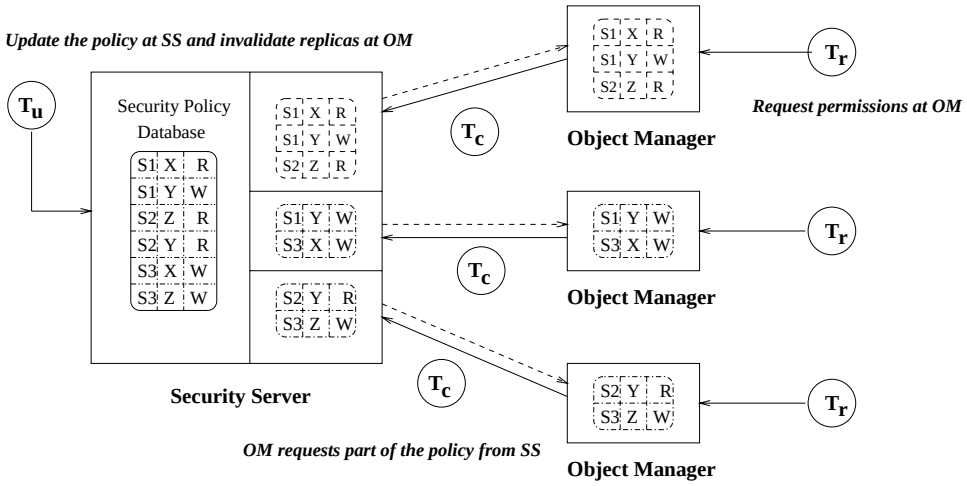

Fig. 4.3. Proposed System Model

The model is composed of three types of transactions as illustrated in Figure 4.3. They are described in turn below.

1. Update transactions T_u are initiated only at the SS. Each update consists of reads and writes in any order. Any replica cached at OMs must be updated or invalidated before a write is complete.

2. A read transaction T_r executes only at the OM. The transaction is complete after it has read all access permissions it needs from the OM's cache. If there is a cache miss, the read transaction aborts and restarts only after the missing cache has been filled.

3. A cache-fill transaction T_c initiates at an OM to fill a missing cache entry. It sends a request to the SS asking for a copy of specific access permissions. The transaction

is complete after the SS sends back the permissions it requests and the cache entry at the OM has been filled.

4.2 Consistency Protocols

A consistency protocol consists of two parts. The first part is a synchronization mechanism for controlling concurrent accesses to replicated data. Most commonly used synchronization methods are two-phase locking and timestamp ordering. The second part is a replica control protocol for controlling how changes at one site are propagated to other sites. These two parts must work cooperatively so that any access to a replica is atomic and synchronized. In addition, the protocol must ensure that the effects of failure and recovery are transparent to all replicas [6].

In the following, we present two invalidation-based consistency protocols: the Invalidation Lock-Based Consistency Protocol (ILBP) and the proposed Invalidation Timestamp-Based Consistency Protocol (ITBP). The performance of both protocols is compared against that of eager replication which is an update-based protocol. All three protocols use different synchronization and replica control. Both the ILBP and eager replication use two-phase locking, but different replica control. The ILBP requires that all replicas be invalidated before the update commits while eager replication requires that all replicas be updated before the update commits. On the contrary, the ILBP and the proposed ITBP have the same replica control as both are invalidation-based protocols, but they use different synchronization control.

4.2.1 Invalidation Lock-Based Consistency Protocol

The Invalidation Lock-Based Consistency Protocol (ILBP) requires that all replicas at OMs must be invalidated before the update transaction at the SS commits. The protocol uses a locking mechanism to synchronize concurrent accesses to the security policy database at the SS and to all replicas at OMs. The protocol also applies a variation of *strict* two-phase locking to all three transactions in the model. A set of message types is defined in the ILBP as follows.

- GET Message. An OM sends a GET message to the SS asking for a copy of access permissions.
- GRANT Message. The SS sends a GRANT message to an OM in response to a GET message from the OM. The message contains a copy of access permissions granted to the OM.
- DENY Message. The SS sends a DENY message to an OM in response to a GET message from the OM. It indicates that the request from the OM has been denied, and the message does not contain a copy of access permissions.
- REVOKE Message. The SS sends a REVOKE message to an OM indicating that the OM must invalidate its cached copy of access permissions.
- NOTIFY Message. The OM, who received a revocation message from the SS and has invalidated its cached copy, sends a NOTIFY message back to the SS.

The three transactions in the ILBP work as follows:

- For an update transaction at the SS, the SS must acquire a lock on the entry representing specific access permissions before doing reads or writes. If it fails to get the lock, the transaction aborts and releases all locks it held. The aborted transaction undoes any modification it made and may restart.

  In case of writes, if there are replicas at some OMs, the SS sends a REVOKE message to those OMs. Upon receiving the message, the OM acquires a write lock on the replica, invalidates the replica, sends a NOTIFY message back to the SS, and releases the lock. The SS must wait until it receives all NOTIFY messages before it can perform the write. The transaction releases all locks at the SS after it commits.

- For each read in a read transaction at an OM, if access permissions regarding the designated object and requested subject are cached, the OM acquires a read lock on the corresponding cache entry before reading it. If there is a cache miss, the transaction aborts. It restarts only after the missing cache entry has been filled. All locks are released after the transaction commits or aborts.

- A cache-fill transaction initiated at an OM acquires a write lock on an available cache entry before sending a GET message to the SS. The SS then acquires a read lock on the entry containing access permissions requested on behalf of the cache-fill transaction. If the SS can get the lock, it reads the entry, sends a GRANT message to the OM, and releases the lock. As a result, the missing cache is filled and the write lock at the OM is released. But, if the SS fails to get the read lock, it sends

a DENY message to the OM. The OM sends another GET message to the SS until a GRANT message is received from the SS.

During the update, the revocation does not need to hold locks at an OM after a replica has been invalidated since the OM will ask for a new copy of access permissions from the SS whether the update commits or aborts. As a result, two-phase commit is not required for update transactions, and there is no deadlock between read and update transactions. However, deadlocks are possible among update transactions at the SS, and they can be detected using a wait-for graph [47].

For better performance, we allow shared read locks among cache-fill transactions from OMs. Also, a cache-fill transaction may be blocked and waits for a lock until the lock is released by the update transaction. The ILBP achieves 1SR because strict two-phase locking is applied for update transactions. Hence, no other transactions can read the updated policy until the transaction performing the update commits or aborts. The proof for 1SR consistency of the ILBP is given in [43].

4.2.2 Invalidation Timestamp-Based Consistency Protocol

In this section, we propose another invalidation-based consistency protocol which uses timestamp ordering as synchronization control. The protocol is motivated by two goals: (i) to relax the coordination between the SS and OMs while achieving 1SR consistency, and (ii) to avoid deadlocks at the SS. The description of timestamp ordering is given in Section 2.1.2.2. The proposed protocol uses *strict* timestamp ordering since it requires that uncommitted data cannot be read until the transaction performing the update commits or aborts. As a result, the proposed protocol ensures 1SR consistency

because of strict execution given by *strict* timestamp ordering, and because of unique and totally ordered timestamps generated at all sites. We explain how timestamps are generated at each site to ensure their total order and uniqueness in the following section.

4.2.2.1 Generating Timestamps

In a distributed system, an ordered pair of two numbers is usually used as a timestamp. The first number is simply a counter that is incremented every time a new timestamp is generated. The system clock may be used as it always increases. The second number is a unique number assigned to each site. Its IP address is commonly used.

To ensure serializability, timestamps generated at all sites must be unique and totally ordered since a timestamp assigned to a transaction initiated at one site is also used to access a replica at another site. In our protocol, a timestamp at the SS is increased whenever there is a new update, and an OM can use the update timestamp it receives in a message from the SS to generate its local timestamps.

A timestamp generated in our protocol is an ordered triplet of numbers, say, (S1, S2, S3) at the SS, and (O1, O2, O3) at each OM. At the SS, the value of S1 is initially zero and incremented by one whenever the SS creates a new update transaction. The value of S2 is always zero since only update transactions are created at the SS, and the value of S3 is the IP address of the SS.

At an OM, the value of O1 is initially zero when it starts making a request to the SS. It is set to the value of S1 received in a message from the SS if the value of S1 is greater than its current value. The value of O2 is initialized to zero and incremented by

one whenever a new transaction is created at the OM. However, when the value of O1 is updated to S1, the value of O2 is set back to zero. The value of O3 is the IP address of each OM. For the purpose of simulation, a single number is simply assigned as an IP address of each site. That is, S3 is zero and O3 is uniquely assigned in turn from 1 to the number of OMs.

Every access permission x at the SS and its replica at OMs is also tagged with $max\_read\_ts[x]$ and $max\_write\_ts[x]$. Both timestamps are updated only by the latest read and the latest write. Moreover, the protocol uses *strict* timestamp ordering [6] to synchronize concurrent accesses and to ensure 1SR consistency since it requires that uncommitted policy updates cannot be read until the update transaction commits or aborts.

4.2.2.2 Description of the Proposed Protocol

In this protocol, a timestamp is generated at the SS and at OMs as (S1, S2, S3) and (O1, O2, O3), respectively. When a new transaction is created at each site, the corresponding timestamp, newly generated, is tagged to that transaction. When a transaction sends a message to another site, its timestamp is also tagged to the message so that it can be used to access replicas at that site. Thus, the body of a message is a tuple of the form (*subject-id, object-id, access-permissions, message-type, sender, timestamp*). Six message types are used in this protocol. They include all five message types defined in the ILBP, and have an additional NACK message type as a reply from an OM to the SS when a revocation from the SS is rejected. The following describes

how the three transactions work in our protocol. We assume that x designates access permissions that a transaction is requesting, either read or write.

- For every timestamp generated at the SS, S2 and S3 are always zero. Thus, each update transaction is uniquely identified by the value of S1. When an update transaction is created, S1 is merely incremented and concatenated with S2 and S3 to form a new timestamp. The timestamp is then tagged to every read and write in the update transaction. According to timestamp ordering rules, a read or write to x is executed if its timestamp is equal or greater than $max\_read\_ts[x]$ or $max\_write\_ts[x]$, respectively. Otherwise, it is rejected.

  If there are replicas of x at some OMs, the update transaction sends a REVOKE message tagged with its timestamp to those OMs. An OM, who receives the message, schedules it as a write. Since S1 in the message is likely higher than O1 at the OM, the revocation is executed and O1 is updated to S1. The OM then invalidates its replica of x and sends a NOTIFY message back to the SS.

  The write to x in the update transaction is complete after it received all NOTIFY messages from OMs. However, if the revocation is rejected, the OM will send a NACK message to the SS. As a result, the update transaction aborts and restarts with a new and higher timestamp.

- For a read transaction at an OM, a new timestamp denoted as (O1, O2, O3) is assigned to the transaction. Each timestamp generated for read transactions is mostly differentiated by the value of O2 since it is a running number and O1 can be changed only by revocation from the SS or by cache-fill transactions. The

timestamp is then tagged to each read in the transaction. If x is in the OM's cache and the timestamp of a read is equal or higher than $max\_write\_ts[x]$, the read is executed. Otherwise, the transaction aborts. If x is not in the OM's cache, the transaction restarts with a new timestamp only after x is in the cache by a cache-fill transaction. The transaction is complete after it reads all x in the OM's cache.

- Since a cache-fill transaction is created at an OM, it is assigned a new timestamp generated as (O1, O2, O3) in the same way as read transactions. Also, since a cache-fill transaction is created only after x is revoked, O1 is just updated to S1 and O2 is set to zero for its timestamp. After searching for an available cache entry, the transaction updates $max\_write\_ts[x]$ at that entry and starts executing by sending a GET message to the SS. The SS regards the message as a read and uses the timestamp in the message when scheduling it.

  If the timestamp of the GET message is higher than $max\_write\_ts[x]$ at the SS, a GRANT message containing the modified value of x and $max\_write\_ts[x]$ is sent to the OM. The OM then copies x to its cache entry. Since $max\_write\_ts[x]$ in the GRANT message indicates the latest update of x, the OM updates its O1 to S1 in $max\_write\_ts[x]$, and set its O2 to zero. Then, the transaction is complete.

  However, if the timestamp in the GET message is less than $max\_write\_ts[x]$, a DENY message containing $max\_write\_ts[x]$ is sent to the OM. As a result, the OM updates its O1 to the new S1 before aborting the transaction. Next, a new cache-fill transaction restarts with a new timestamp generated using the updated

O1. It then sends another GET message to the SS. Unless the OM receives a GRANT message and has a copy of x in its cache, a new transaction will restart.

Note that when there is only one update transaction created at a time, its read and write are never rejected and the transaction is not aborted since S1 is always higher than O1 in timestamps of cache-fill transactions. In addition, its revocation is never rejected at OMs for the same reason. However, when there are more than one update transaction concurrently executed at the SS, a read or write of one update transaction to x may be rejected if $max\_read\_ts[x]$ or $max\_write\_ts[x]$ is updated by another update transaction. This will also have a similar effect to the value of O1 at OMs. As a result, a cache-fill transaction may restart several times before x is replicated in the OM's cache.

One benefit of this protocol is that there is no deadlock at the SS since any late read or write will be rejected and cause its transaction to abort. Also, reads can be shared among update and cache-fill transactions and result in lower response time for cache-fill transactions. However, cache-fill transactions may restart more often than those of the ILBP since a higher timestamp is required for a new cache-fill transaction so that it would not be rejected again.

4.3 Performance Evaluation

We evaluate the performance of the two invalidation-based protocols and compare them with that of eager replication since it also gives 1SR consistency. The description and the implementation of eager replication for our model is given in this section. We simulate the system model and implement the three transactions of the model for each

protocol. Our simulation is based on transaction processing models used in a typical database system [19].

4.3.1 Eager Replication

Eager replication is an update-based consistency protocol requiring that all replicas at OMs must be updated before the update commits at the SS. The protocol uses locking mechanism to control concurrent accesses to the policy database at the SS and to every replica at OMs. Strict two-phase locking is applied to all transactions in the model.

In addition to GET, GRANT and DENY messages defined in the ILBP, the protocol requires additional message types for its replica control as follows.

- UPDATE Message. The SS sends an UPDATE message to OMs to ask OMs to update their replicas.
- ACK Message. If an OM successfully updates its replica, it sends an ACK message back to the SS.
- NACK Message. If an OM fails to update its replica, it sends a NACK message back to the SS.
- COMMIT Message. The SS sends a COMMIT message to an OM so that the OM can commit its update.
- ABORT Message. The SS sends an ABORT message to an OM so that the OM aborts and undoes its update.

- CACK Message. The OM, who had received a COMMIT or an ABORT message from the SS, sends a CACK message back to the SS.

Read transactions and cache-fill transactions in this protocol work the same way as those in the ILBP. However, its update transactions work differently and they are described below.

- An update transaction must acquire locks for each read and write in the transaction. In case of a write, after getting the lock at the SS, an UPDATE message is sent to all replicas at OMs. If an OM can get the lock, it updates its replica and sends an ACK message back to the SS. Otherwise, the OM sends a NACK message to the SS. After the SS receives ACKs for all writes, the transaction is ready to commit by sending a COMMIT message to OMs. An OM, who receives the message, now can release the lock and sends a CACK message back to the SS. The update transaction can commit only after it receives all CACKs from OMs.

  If the SS receives only some ACKs, the transaction must abort. It then sends an ABORT message to OMs including those who already updated their replicas. Upon receiving the message, an OM must undo its update, release the lock and send a CACK message back to the SS. After receiving all CACKs from OMs, the transaction aborts.

  In case that a write in the transaction fails to get a lock at the SS, the transaction must also abort. The transaction must send an ABORT message to OMs and wait for all CACKs from OMs before it releases all locks at the SS.

At OMs, there are direct contentions between an update transaction from the SS and read transactions at OMs since an update transaction requires to hold locks at some OMs while trying to obtain locks held by read transactions at other OMs. If the SS wants to update part of the policy that is often cached at some OMs, a chance of obtaining locks at all OMs is small. As a result, the possibility of having a deadlock between read and update transactions is high. Even though a deadlock can be avoided using a *time_out*, it would be better if a deadlock can be detected and resolved since selecting a proper *time_out* is not easy. In our simulation, we implement a distributed deadlock detection for this protocol. As a result, a read or a write may be blocked while waiting for a lock unless its blocking causes a deadlock. Note that the cost of deadlock detection is usually expensive since it requires extra messages communicating among sites to maintain a wait-for-graph.

4.3.2 Workload

The workload to the model is given by two types of transactions at two places. Update transactions consist of reads and writes and are initiated only at the SS. A read transaction consisting of reads is initiated at each OM. The number of operations in a transaction is called *transaction size*. The number of writes in an update transaction is computed using *write ratio*. For instance, if *write ratio* is 0.8, a transaction consists of 80% writes and 20% reads and they can be in any order.

We assume that one entry in the security policy represents a security decision which is a set of access permissions given to a subject to access an object. Each operation in a transaction specifies which entry it wants to read or write. We also assume that one

part of the database called a *hot spot* is accessed more often than the other part. The parameter *access_skew* specifies the probability that an entry in the *hot spot* is accessed. *Think_time* determines the time interval between two update transactions generated. Since there is only one read transaction executing at a time at one OM, the number of OMs (*num_om*) also indicates the number of read transactions. The description of all workload parameters are shown in Table 4.1.

Table 4.1. Workload Parameters

db_size	the number of elements in the security policy database
write_ratio	a ratio of write operations in an update transaction
access_skew	probability that a hot spot is accessed
read_size	read transaction size
update_size	update transaction size
num_om	the number of OMs (read transactions)
num_mpl	the number of concurrent update transactions
Think_time	time between update transactions

Table 4.2 shows system parameters in the simulation as they indicate the CPU time and I/O time used for activities during the execution of a transaction. We assume that each site has a single CPU. Hence, one site can execute only one activity at a time. Since the SS and OMs are on different machines, activities on the SS and at each OM can run in parallel. We also assume that all machines share a common medium when they are communicating and only one machine can send a message at a time. The average time used to send a message between machines is given by *Comm_time* and the average time used to communicate between processes within a machine is given by *Delay_time*.

Table 4.2. System Parameters

Comm_time	communication time between SS and OM
Delay_time	communication time within SS or OM
IO_time	time to write a log of updates
ReadSS_time	time to read entry at SS
UpdateSS_time	time to update entry at SS
ReadCache_time	time to read entry at OM
UpdateCache_time	time to update entry at OM
RevokeCache_time	time to invalidate entry at OM

In our model, there should be some overhead for obtaining locks or generating timestamps. However, we assume that the overhead is insignificant, and we would expect that performance differences among the protocols should come from the way each protocol keeps all replicas consistent rather than from the cost of concurrency control itself.

4.3.3 Queueing Model

Our simulation uses the closed queueing model as shown in Figure 4.4. There are a fixed number of processes from which transactions originate. For update transactions, there are *num_mpl* of *update processes*. For read transactions, there is only *one read process* per OM. The model consists of four types of processes described in turn below.

- An *update process* generates an update transaction and sends each operation to the SS one at a time. It is also responsible for releasing all locks at the SS after the transaction commits or aborts. When a transaction ends, a new transaction is generated after *Think_time*.

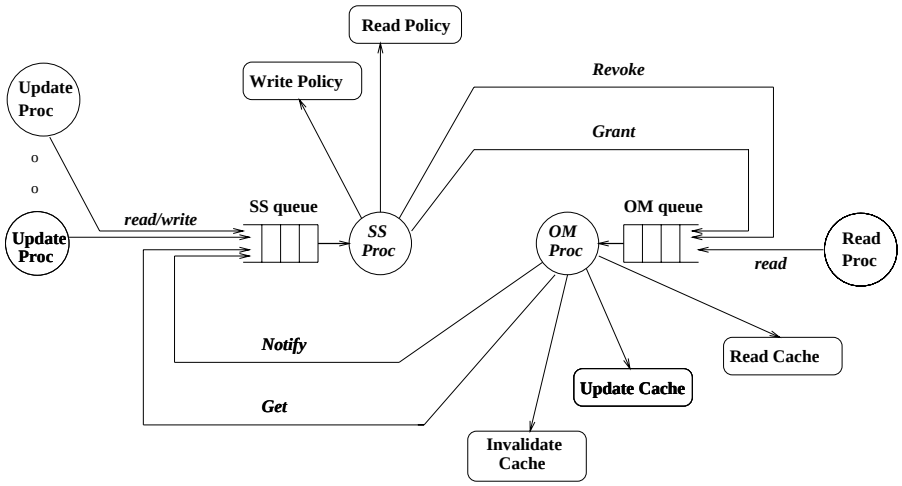

Fig. 4.4. Logical Queueing Model

- A *read process* generates a read transaction and sends each read to an OM to read a security decision cached at the OM. It is also responsible for releasing all read locks after the transaction commits or aborts. When a transaction ends, a new transaction is generated immediately.

- The *SS process* is responsible for synchronizing any access to a security decision at the SS by update transactions or by cache-fill transactions.

- The *OM process* is responsible for synchronizing any access to a cached copy of a security decision at an OM by read transactions or by a revocation from update transactions. It is also responsible for creating a cache-fill transaction to fill in the missing cache.

4.3.4 Performance Measurement

In our simulation, the performance is measured at two places: update transactions at the SS and read transactions at OMs. It can be measured in terms of transaction throughput (the number of transactions completed in a second) and transaction response time (the time taken to complete a transaction). Since a transaction may be aborted due to various factors, we assume that an aborted transaction is restarted until it is complete. As a result, the response time measured includes the time the transaction had aborted. In addition, since these two performance metrics are typically inversed, only transaction response time is reported.

Intuitively, the performance of a transaction is directly related to the transaction size. That is, when the transaction size increases, the response time would increase. But, it may be affected by conflict accesses from other transactions as well. In both the ILBP and the ITBP protocols, the update response time also depends on the time taken for revocations at OMs. It may be delayed by cache-fill transactions initiated earlier by OMs. In eager replication, an update transaction has direct contention with read transactions at OMs that may lead to deadlocks and cause high aborted rate of update transactions. Thus, the response time may be high.

Furthermore, the read response time in both the ILBP and the ITBP protocols depends directly on the response time of cache-fill transactions. More revocations from update transactions also cause higher aborted rate of read transactions and, hence, make read response time increase. Similarly, in eager replication, the read response time may be delayed by the time waiting for locks to be released by update transactions. However,

after update transactions abort or commit, a read transaction can continue or restart immediately since the updated security decisions have been placed locally at OMs. Thus, its read response time may be lower than those of both invalidation-based protocols.

4.4 Experiments and Results

To compare the performance of the three protocols, we design three experiments. Experiment 1 studies the effect of varying update transaction size assuming that we have only one update transaction and a fixed number of OMs. Experiment 2 investigates the effect of varying the number of OMs. Experiment 3 examines the relative performance of the three protocols when there are many update transactions running at the SS concurrently.

Table 4.3 shows the range of values of workload and system parameters for all experiments. We assume that all time parameters have the same time unit in the simulation. The database size (*db_size*) is chosen to be 400 entries. It is small in size to allow significant data contention among transactions.

In each experiment, we have three classes of workload we call: *heavy*, *moderate* and *light*. These workloads provide high, moderate, and low data contention among update and read transactions, respectively. The values of workload parameters for each class are given in Table 4.3. We assume that all OMs have *full* replication when the simulation starts. In other words, all elements in the security policy database are initially cached at every OM. We also assume that the cost of reading and writing each entry at the SS and at OMs is approximately equal and is set to be 0.001 time unit. The I/O time only accounts for writing committed update transactions to a log file as usually implemented

Table 4.3. Parameters settings

Workload parameter	*heavy*	*moderate*	*light*
db_size	400	400	400
write_ratio	0.8	0.5	0.3
access_skew	0.7	0.5	0.5
Update_size	10-50	10-50	10-50
Read_size	10	10	10
num_om	1-20	1-20	1-20
num_mpl	1-15	1-15	1-15
Think_time	0	2	5
System parameter	time unit		
Comm_time	0.005		
Delay_time	0.001		
ReadSS_time	0.001		
UpdateSS_time	0.001		
ReadCache_time	0.001		
UpdateCache_time	0.001		
RevokeCache_time	0.001		
IO_time	0.005		

in a traditional database. Each data reported is the average of results of 20 runs where each run lasts for 4000 time units.

In addition, since the performance of the two invalidation-based protocols is comparable and distinguishable from that of eager replication, each experiment reports two sets of results. One is the results of the ILBP versus eager replication, and the other is the results of the ILBP protocol versus the ITBP protocol.

4.4.1 Experiment 1 : Varying update transaction size

In this experiment, the update transaction size is varied. The results in Figure 4.5 show that update transactions in the ILBP perform much better than those of eager replication for all update workload. As can be seen, increasing update transaction size causes a slight increase to the response time of the transaction in the ILBP, but causes an enormous increase in the case of eager replication. The main reason is that the aborted rate of update transactions in eager replication due to deadlocks is very high, whereas there is no deadlock in the ILBP.

Figure 4.6 shows that read transactions in eager replication have better performance than those of the ILBP. That is, with increasing update transaction size, read transactions in the ILBP have higher response time than those of eager replication. This is because higher update workload leads to more revocations at OMs and results in higher response time in cache-fill transactions as they have to wait longer for the locks to be released by update transactions. In contrast, read transactions in eager replication can complete locally at OMs. Hence, aborted update transactions have less effect on read

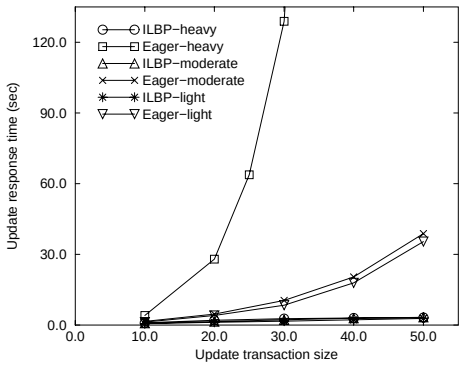

Fig. 4.5. Update Transaction Response Time

transactions. Note that for *light* update workload, read response time of the ILBP is comparable to that of eager replication.

The results in Figure 4.7 show that the update response time in the ILBP is slightly lower than that of the ITBP for all update workload. This is because the update transaction in the ITBP has higher data contention with cache-fill transactions than that in the ILBP as the ITBP protocol allows shared reads between cache-fill transactions and update transactions while the ILBP protocol allows shared reads among cache-fill transactions only.

Furthermore, when update transaction size increases, increasing rate of update response time in both protocols is approximately linear for *moderate* and *light* update workload. But, for *heavy* update workload, increasing rate of update response time is slower since there is less data contention from cache-fill transactions as most of them are now waiting for the update transaction to commit.

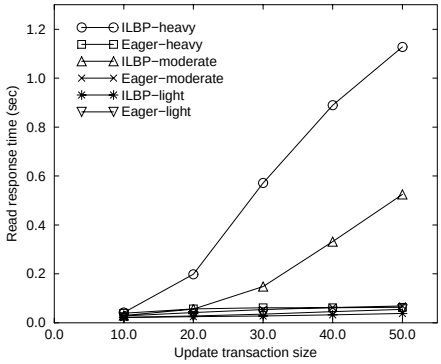

Fig. 4.6. Read Transaction Response Time

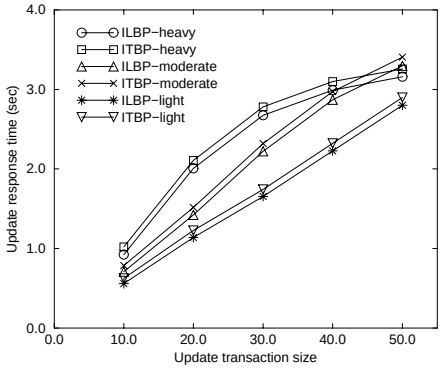

Fig. 4.7. Update Transaction Response Time

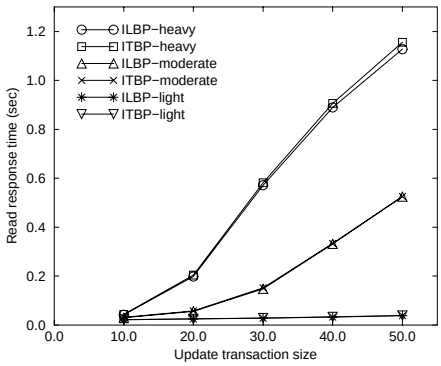

Fig. 4.8. Read Transaction Response Time

Figure 4.8 shows that read response time of both protocols are comparable and increases when update transaction size increases. In addition, the rate of increase in read response time is higher with increasing update workload due to higher data contention of cache-fill transactions with those updates.

4.4.2 Experiment 2 : Varying the number of OMs

Figure 4.9 and Figure 4.10 illustrate similar results to those in Experiment 1. Figure 4.9 shows that the update response time of the ILBP is slightly higher when the number of OMs increases. This is due to an increase in the response time of cache-fill transactions. However, in eager replication, the update response time is much higher with increasing number of OMs since the aborted rate of update transactions is higher. With less update workload, the update response time decreases accordingly.

The results shown in Figure 4.10 indicate that, for *heavy* update workload, increasing the number of OMs has a larger effect on read response time in the ILBP than

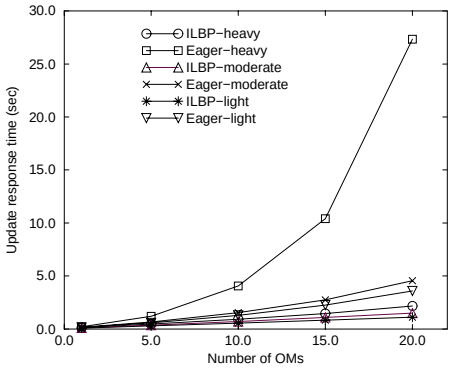

Fig. 4.9. Update Transaction Response Time

that in eager replication. This is because increasing the number of OMs also increases the response time of cache-fill transactions due to the communication delay between the SS and OMs. Note that all sites share a common medium for communication. In contrast, read transactions in eager replication can complete locally at OMs.

Figure 4.11 shows that the update response time of both invalidation-based protocols almost linearly increases when the number of OMs increases. But, at the same number of OMs, the update response time of the ITBP is higher than that of the ILBP because cache-fill transactions in the ITBP have higher data contention with update transactions than those in the ILBP.

The results in Figure 4.12 illustrate that read response time of both invalidation-based protocols increases with increasing number of OMs due to higher response time of cache-fill transactions. Read response time is also higher with increasing update workload. However, read response time of the ILBP is a little lower than that of the ITBP because more than one request may be sent to the SS for one GRANT message

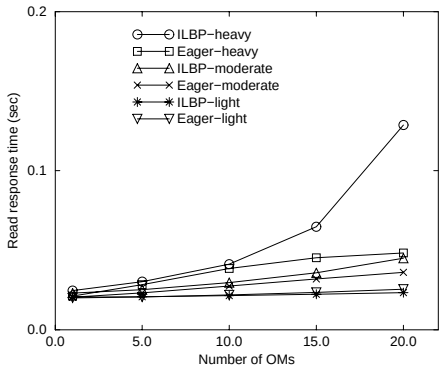

Fig. 4.10. Read Transaction Response Time

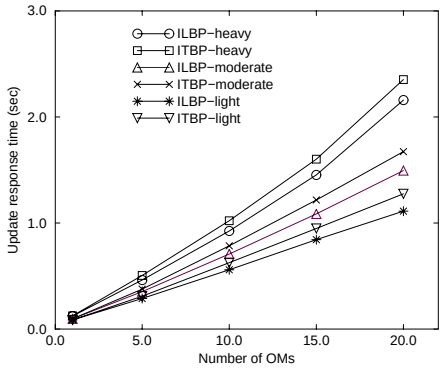

Fig. 4.11. Update Transaction Response Time

while only one request is sent in the ILBP. Hence, cache-fill transactions in the ITBP have higher response time.

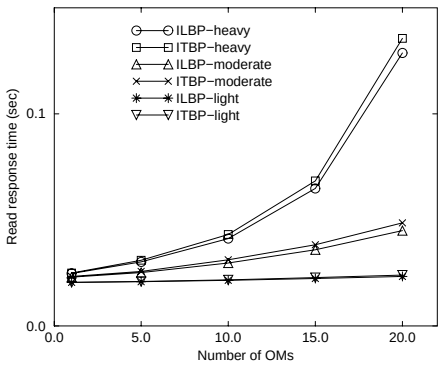

Fig. 4.12. Read Transaction Response Time

4.4.3 Experiment 3 : Varying the number of update transactions

In this experiment, the number of concurrent update transactions at the SS is varied while the update size and the number of OMs are fixed. Since there are more than one update transaction executing concurrently at the SS, deadlocks may occur not only between read and update transactions, but among update transactions themselves as well. As a result, the response time of both read and update transactions is higher than those in Experiment 1 and Experiment 2.

The results in Figure 4.13 indicate that, for all update workload, the update response time in eager replication is much higher than that of the ILBP because of higher aborted rate of update transactions.

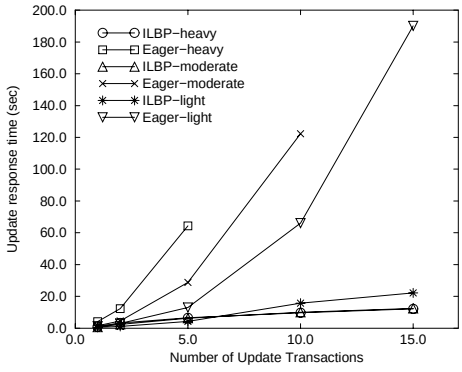

Fig. 4.13. Update Transaction Response Time

The results in Figure 4.14 show that, for all update workload, when there are a few update transactions, read response time of eager replication is lower than that of the ILBP since the higher aborted rate of update transactions in eager replication allows higher completion rate of read transactions. But, when there are more update transactions holding locks at OMs, they cause more deadlocks and lower completion rate of read transactions. Thus, its read response time increases significantly. For the ILBP, deadlocks occur among update transactions at the SS only. Hence, its read response time is delayed only by the response time of cache-fill transactions.

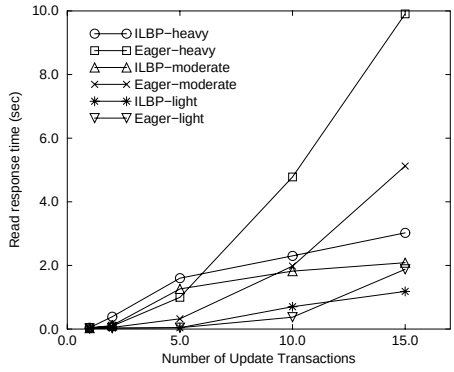

Fig. 4.14. Read Transaction Response Time

Figure 4.15 illustrates that, for all update workload, update response time of both invalidation-based protocols increases with increasing number of update transactions. When there are a few updates, less data contention among updates leads to lower update response time for lower degree of update workload. However, with higher number of updates, update response time for *light* workload is higher than that of *heavy* and *moderate* since higher number of reads in updates causes higher data contention with cache-fill transactions.

Moreover, for *heavy* workload, the update response time of the ITBP is significantly higher than that of the ILBP because the ITBP aborts updates to ensure serializability whereas the ILBP aborts updates only to resolve deadlocks. Hence, the ITBP protocol has much higher aborted rate of updates. But, for *moderate* and *light* workload, the update response time of both protocols are comparable since the aborted rate of updates of both protocols are lower. Reducing the degree of update workload has a larger effect on the aborted rate of updates in the ITBP protocol than in the ILBP protocol.

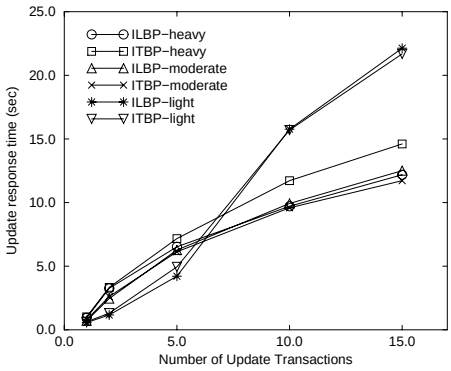

Fig. 4.15. Update Transaction Response Time

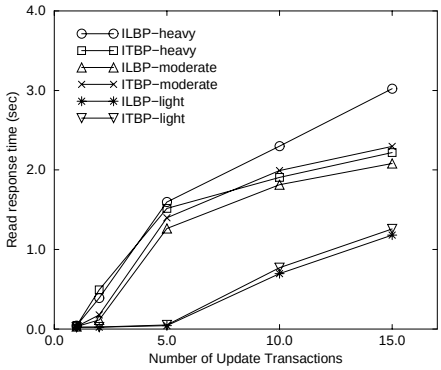

Fig. 4.16. Read Transaction Response Time

The results in Figure 4.16 indicate that a lower degree of update workload leads to lower read response time as most reads can complete locally at OMs. In addition, for *heavy* workload, the read response time of the ITBP is lower than that of the ILBP because the ITBP protocol allows shared reads among updates and cache-fill transactions resulting in lower response time of cache-fill transactions. However, for *moderate* and *light* workload, the read response time of the ITBP is higher than that of the ILBP due to higher delay in the response time of cache-fill transactions.

4.5 Related Work

Most work for maintaining *data consistency* focus on update propagation protocols such as eager replication and lazy replication [7, 24]. While eager replication gives 1SR consistency, its performance drops drastically when the number of sites increases [24]. Lazy replication offers higher performance, but at the expense of weak consistency. Recent work on lazy replication has been presented in [3, 9, 15].

The problem of maintaining *authorization consistency* in distributed databases has been addressed in [41]. The authors present an authorization model that allows transient inconsistency as changes in authorizations can be initiated asynchronously and possibly simultaneously at several sites. The model is based on lazy replication and requires that changes be committed at one site before being propagated to other sites. As a result, different sites may receive changes in different order. To achieve consistency, a set of algorithms is specified for each authorization operation such as *grant* and *revoke* to reconcile any inconsistent authorization. Each site must maintain logs of messages it

receives so that the algorithm running at one site can consult those logs for reconcilation. However, the algorithms do not attempt to resolve any possible conflict.

Sequence Number Synchronization Protocol (SNSP) presented in [43] maintains consistency of the policy between the SS and the OM using invalidation, but it imposes no notion of transactions. The protocol assumes that communication between the SS and the OM is not reliable, and messages sent between them may be lost or arrive out of order. Thus, a sequence number is tagged to every message in order to determine the relative order of messages sent. If a message arrives out of order, the receiver can decide to ignore or accept the message. In addition, there is no concurrency control imposed at both sites.

4.6 Chapter Conclusion

In this chapter, we presented two consistency protocols: Invalidation Lock-Based Consistency Protocol (ILBP) and our proposed Invalidation Timestamp-Based Consistency Protocol (ITBP). Both protocols are invalidation-based consistency protocols and guarantee 1SR consistency. We compare the performance of both protocols with that of eager replication which is an update-based consistency protocol also ensuring 1SR consistency.

The system we consider is composed of the Security Server and multiple Object Managers located on different machines. For the sake of fairness, we compared the performance of the ILBP with that of eager replication since both have the same synchronization control, but different replica control. Then, we compared the performance

of the ILBP with that of the ITBP because they have the same replica control, but different synchronization control.

The simulation results show that invalidation-based protocols give better update performance than the update-based protocol for all update workload. The main reason is due to the fact that invalidation-based protocols do not require two-phase commit and, hence, there is no deadlock between update and read transactions. In addition, no rollback at OMs is required if updates abort. The results also illustrate that all protocols offer comparable read performance except under *heavy* update workload when most reads in eager replication can complete locally at OMs, whereas reads in invalidation-based protocols are aborted more often due to more revocation from the SS.

In addition, the results show that the ILBP offers slightly better update performance than the proposed ITBP protocol for all update workload since the ILBP has lower aborted update rate than the proposed ITBP protocol as its update transactions abort only to resolve deadlocks. However, both protocols give comparable read performance. We conclude that invalidation-based protocols offer good performance for policy reading as well as for policy update.

Moreover, in invalidation-based protocols, an OM may retain a replica of the security policy or not at all, whereas, in the update-based protocol, an OM may retain either an old replica or a new replica of the policy. As a result, the restriction of accessing the policy during the update is more effective in invalidation-based protocols than that in the update-based protocol. Currently we assume that there is no site or communication failure. However, site failure is simple to manipulate by invalidating since any recovered

OM must send new requests to the SS. Also, the SS may keep track of which OM has failed.

In this system model, changing the security policy only at the SS may cause a bottleneck if there are many requests for updating the policy. However, we would expect that changes in the security policy do not happen very often. To alleviate this bottleneck, we may have the SS residing on a more powerful machine or allow the security policy to be partitioned and distributed across several machines. The latter would require that only one machine maintains the *primary copy* of a specific portion of the policy and any update to the portion must be initiated only on that machine.

A final observation is that, in implementing a security server, the cost of synchronization control cannot be ignored. In addition, two-phase locking and deadlock detection may complicate the implementation. The proposed ITBP protocol is an alternative since it offers both strong consistency and comparable performance.

Chapter 5

Authorization Model for Summary Schema Model

Unlike a typical homogeneous distributed database, a multidatabase system (MDBS) integrates existing and possibly heterogeneous databases while preserving their local autonomy [44]. The motivation is to join individual and independently developed databases without any modification and hence to save the existing investment. The MDBS will act as a front end to local databases and facilitate any operation performed across them.

Local autonomy in an MDBS allows local databases to maintain complete control over their local resources [44]. Local autonomy comes in the form of communication autonomy, execution autonomy, design autonomy, and authorization autonomy. In other words, after joining the global information sharing environment, an individual local database can continue to maintain its current data model and query language. It can decide to globally share its local resources. Moreover, it should be able to execute its local operations without interference from external sources. Finally, *authorization autonomy* [17] allows a local database to accept or reject any request from others that accesses its data.

Due to local autonomy, each local database can join or depart an MDBS at any time without major changes. It simply adds global functions to access the MDBS while its local functions remains unchanged. However, each database joining an MDBS may be heterogeneous. Specifically, they often have different data models and different

query languages. The MDBS must be able to map various data models to a common and canonical model. In addition, semantically similar data may have different names and representations while different data may have identical names. The MDBS must resolve these differences and similarities based on the semantics of data. These overheads may degrade global performance significantly as the MDBS has no control over local databases.

There are two general approaches to resolving multidatabase heterogeneity [12]. In the first approach, local schemas are integrated to form a global schema representing common data semantics. The process of integration is rather complicated, labor intensive, and probably requires manual manipulation. The global schema is usually maintained at every site to allow simple and fast accesses to the data. Duplication of the global schema, however, raises the consistency problem in the case of updates at local databases [10].

In the second approach, the integration of local schemas is achieved through a common multidatabase language which interprets and transforms a query to data represented and maintained at local databases. There is no global view of shared data. Thus, a user must know the location and the representation of data being queried. The approach is naturally less transparent than the global schema approach, but it is more efficient and less complicated.

The Summary Schemas Model (SSM) [11] is proposed as an adjunct to multidatabase language systems for supporting the identification of semantically similar data

entities. The model resolves name differences using word relationships defined in a standard thesaurus such as Roget's Thesaurus. It builds a hierarchical structure of metadata based on access terms exported from underlying local databases.

While most research in multidatabases focus on update issues [10], consistency and concurrency control [29], and query optimization [33], security issues in multidatabases have received little attention. A number of authorization models for multidatabases have been proposed [13, 17, 36, 48, 49] which mainly focused on authorization models for a system based on global schema integration approach.

In a multidatabase environment, authorization models of local databases must be preserved because of local autonomy. With the addition of heterogeneity, enforcing a single authorization model globally in an MDBS is quite a challenge. One approach is to derive a global authorization model from underlying local authorizations of local databases [13]. Authorizations can be derived for integrated or imported objects based on the similarity between subjects. However, subjects among local databases are unlikely to be compatible and may have conflicting access authorizations to the same object. As a result, no global authorization can be derived for those subjects. Another approach is to propagate defined global authorizations to local databases when local data accesses are requested [26]. However, local databases may accept or reject any global request.

In this chapter, we propose an authorization model for the SSM. The model proposed is based on role-based access control (RBAC) [42]. The motivation is to define a global authorization model that not only is independent of local authorizations, but also inherits common entities which individual local authorization is mapped onto without changing local authorizations. We consider a role as a common representative for users

or subjects in local databases since a role represents a job function defined by an organization where local databases and the MDBS operate. Each local database may map some of its local subjects to a global role defined at the MDBS level. In addition, no local subject identification is maintained at the global level.

Due to the hierarchical structure of the SSM, when more general access terms are formed at a higher level, less degree of authorization is required for accessing those terms. Hence, a global authorization model should be expressed in a hierarchical form. In RBAC, roles can form a role hierarchy and may suite the hierarchical structure of the SSM.

There are two major motivations in enhancing the SSM model by an authorization policy and, hence, limiting accesses to access terms in the SSM hierarchy: (i) Each term has its own degree of sensitivity and should not be accessed by unauthorized subjects. For example, "salary of an employee" in a company should not be publicly accessible while "name of an employee" may be publicly accessible, (ii) any unauthorized access detected as early as possible reduces network traffic and computation which result in increasing the query bandwidth.

Imposing security structure on top of the SSM adds both time and space overhead to the system. It obviously affects the performance of the SSM. The goal of this work is to show the feasibility of such an attempt and to evaluate the performance gain of the enhanced SSM. The remainder of this chapter is organized as follows. First, we give background on role-based access control and role hierarchy. Second, we propose the authorization model for the SSM. Third, we describe the simulation model and experiments used to evaluate the performance of the enhanced SSM and the original

SSM. The experimental results are also discussed. Fourth, we describe related work on authorization models for multidatabases. Finally, we conclude our work in this chapter.

5.1 Background

Role-Based Access Control (RBAC) model [42] is based on three sets of entities: *users*, *roles* and *access permissions*. A role represents a job function in an organization and embodies a specific set of authorizations and responsibilities for the job. In practice, a user is created when a person has a role in an organization. As a member of a role, the user also inherits the same privileges to access objects accessible by the role. If a person no longer works for the organization, the corresponding user must be deleted and his/her privileges are revoked by removing his/her membership from the role.

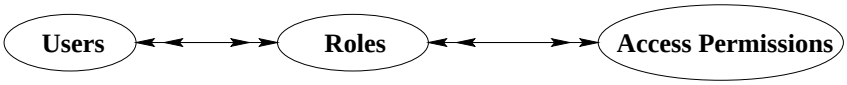

Fig. 5.1. RBAC Many-to-Many Relations

The RBAC model as described represents a many-to-many relation as shown in Figure 5.1. In the model, a user can be a member of many roles and a role can be assigned to many users. Also, a role can have many access permissions and one permission can be assigned to many roles. Furthermore, roles can form a role hierarchy to reflect lines of authority and responsibility in an organization (chain of commands). Thus, a role can

inherit permissions assigned to another role in a role hierarchy. But, some pairs of roles in a hierarchy may be incomparable. Three different forms of role hierarchies [31] are:

- *isa role hierarchy.* This hierarchy is based on generalization. Roles are defined by qualification where one role is more specific than its *isa* role and the role inherits all permissions assigned to its *isa* role and also has its own extra permissions.

- *Activity role hierarchy.* This hierarchy is based on aggregation where a role is defined as an aggregation of its component roles. Thus, the role inherits all permissions given to its components.

- *Supervision role hierarchy.* This hierarchy is based on organizational hierarchy of positions where a role at a higher position inherits all permissions of roles at its lower positions.

The corresponding examples of the three role hierarchies are shown in Figure 5.2.

5.2 Authorization Model for the SSM

We propose a global authorization model for Summary Schemas Model based on role-based access control (RBAC). The main idea is to map an individual subject in local databases to a common role defined at MDBS level, and to tag access terms in the SSM hierarchy with a set of roles allowed to access those objects.

5.2.1 Subjects and Objects

The proposed authorization model specifies subjects and objects both at local databases and at MDBS level. At local databases, there are local subjects and local

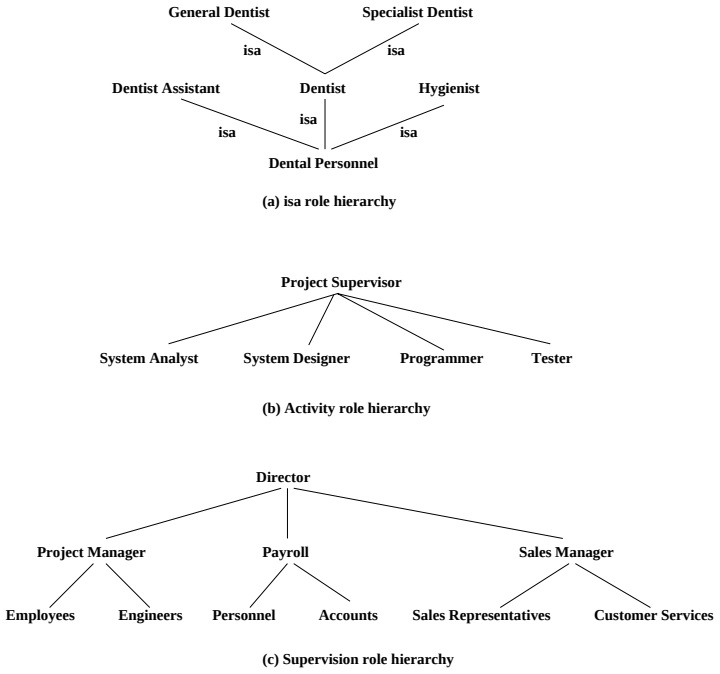

Fig. 5.2. Examples of Role Hierarchies

objects. Local subjects are defined and managed by local databases. Local objects are objects created and maintained at local databases. Each local subject can access its own local objects according to access control rules defined locally and independently at local databases. In addition, we do not make any assumption about authorization models used at local databases.

At MDBS level, each access term in the hierarchy is a global object exported access terms from underlying local schemas. Higher terms are populated in a hierarchical structure according to their word relationships. Thus, there is no composite object at MDBS level. Since global subjects are allowed to access objects across multiple local databases, it is natural to assume that only a subset of local subjects are allowed to be global subjects. Mapping individual local subjects to global subjects is a tedious and error prone task. However, using roles as global subjects simplifies the task. In addition, local databases are responsible for mapping their local subjects to corresponding global roles. Local databases may maintain a table that keeps track of which subject is mapped to which global role. If a new role is added or an existing role is deleted, all local databases will be informed and their local subjects can be remapped. Nevertheless, we do not anticipate frequent changes of global roles. In addition, when a user logs in at any node, the authentication can be done at a local database where a user has an account. No global authentication is needed.

5.2.2 Populating Global Authorizations

When a local database joins the SSM, it registers itself to the SSM so that its semantic contents can be captured in the SSM metadata. The registration process is used

to establish some trust between a joining local database and the SSM. Upon registration, a local database provides the SSM a list of its exported access terms and a list of its local subjects mapped onto global roles. Mapped global roles are then tagged to each exported access term of a node.

When hypernyms of exported access terms are in the SSM hierarchy, a list of authorized global roles is also tagged to those hypernyms. However, since more general SSM access terms are populated at higher level closer to the top of the SSM hierarchy, roles with less privileges are allowed to access those SSM metadata. As a result, if two roles from lower nodes are partially ordered in the role hierarchy defined at MDBS, only the minimum role between the two is tagged to the higher node. But, if two roles are uncomparable, both are tagged to the higher node. Note that if a role has higher privileges than the minimum role at a node, the role does not need to be tagged to the node since it is implicitly allowed to access those terms.

The algorithm for populating authorized SSM nodes is iterative in nature as described in Figure 5.3. The algorithm can be generalized for higher number of local databases and access terms. In addition, there is no assumption about how roles are related in a role hierarchy.

Figure 5.4 and Figure 5.5 illustrate how global authorizations are populated in an SSM node based on the role hierarchy shown in Figure 5.2(a) and Figure 5.2(c), respectively. We briefly explain both examples as follows.

Figure 5.4 shows a hypernym relationship among words related to dental care. Each dental related term is tagged with an authorized role based on the role hierarchy of dental personnel given in Figure 5.2(a). As can be seen that roles "General Dentist"

Assume that we have two local databases. One database exports an access term x accessible to a role r1 and the other exports an access term y accessible to a role r2. An SSM metadata or access term is formed according to the semantic contents of x and y and r1 and r2 role relationship.

(a) If x and y are semantically different, then two SSM nodes are formed as [hypernym of x, r1] and [hypernym of y, r2] at higher level.

(b) If x and y are semantically similar and z is a hypernym of x and y, then we consider r1 and r2 as follows:

(i) If r1 and r2 are partially ordered in the role hierarchy, an SSM node is formed as [z, minimum(r1, r2)].

(ii) If r1 and r2 are not related, an SSM node is formed as [z, r1 or r2].

Fig. 5.3. Algorithm for Populating Global Authorizations

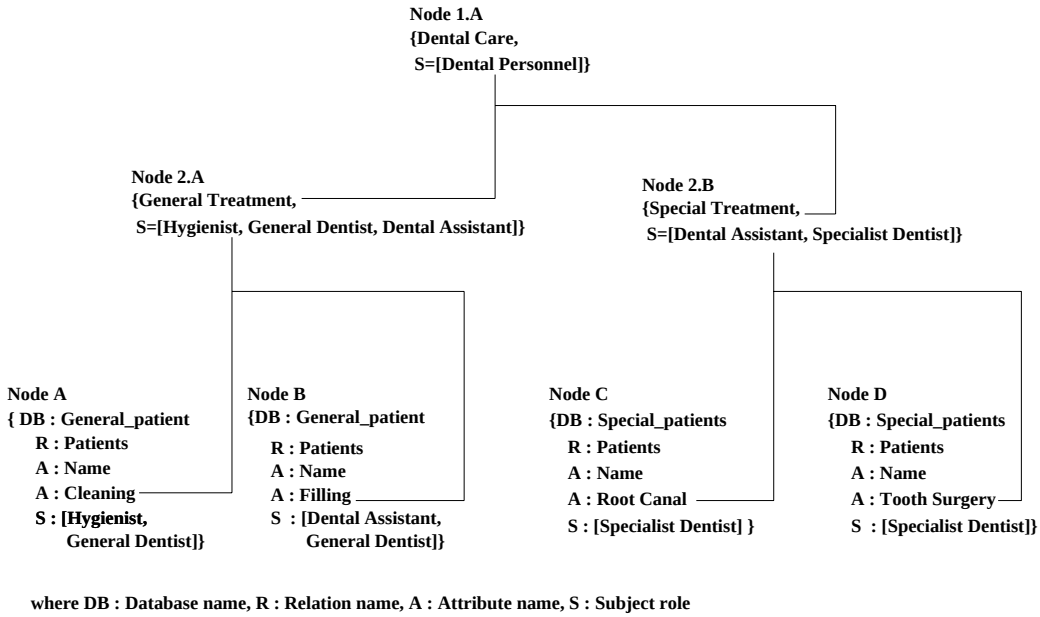

Fig. 5.4. Sample Enhanced SSM Hierarchy with *isa* Role Hierarchy

and "Hygienist" can only access to the term "General Treatment" part of the hierarchy, and cannot access to the term "Special treatment" part.

Figure 5.5 exemplifies a hypernym relationship among words related to personal income of employees in an organization. Each term is tagged with an authorized role based on the hierarchy of employees' role given in Figure 5.2(c). Since, in a supervision role hierarchy, roles at a lower level have less privileges than roles at the higher level, roles tagged to higher terms are the combination of minimum roles tagged at lower terms. For example, at Node 2.A, roles tagged to the node are [Personnel, Employee, Engineer] since they represent all minimum roles at lower nodes (Node A and Node B). The roles [Project Manager] and [Payroll] are implicitly allowed to access the term "Earnings" at Node 2.A, but only the roles [Project Manager] and [Employee] are allowed to access the term "Wage" at Node A. Both roles [Project Manager] and [Payroll] can access the term "Salary" at Node B.

5.2.3 User Query Processing

The algorithm presented in [11] for imprecise query processing is based on the SDM values computed at each node in the SSM hierarchy. Since accesses to terms are now limited to only some authorized roles, the algorithm must be modified so that a query submitted by users of different roles will be properly manipulated according to role privileges of users. As a result, submitting the same query by different roles may not get the same result. The modified query algorithm is described in Figure 5.6.

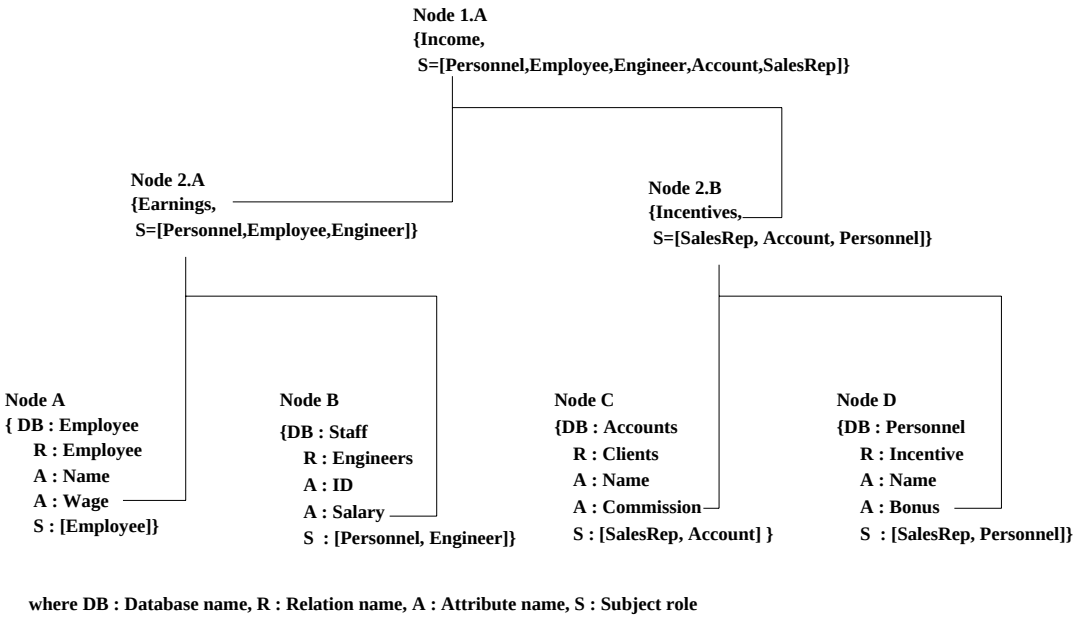

Fig. 5.5. Sample Enhanced SSM Hierarchy with *Supervision* Role Hierarchy

Assumptions:
(1) An imprecise query is submitted at any node in the SSM hierarchy.
(2) If one access term is rejected due to insufficient authority,
the whole query is also rejected.

At query origin node, a query is parsed to identify access terms..
A submitted query is also tagged with a valid global role of the user.

```
1. FOR each imprecise term in the query DO
2.      Compute SDM of each term
3.      IF a match is found and accessible by the user role
4.      THEN IF this is a local node
5.              THEN replace imprecise term with its corresponding precise term
6.              ELSE  send it to lower node and continue at line 2
7.      ELSE IF a match is found, but inaccessible by the user role
8.              THEN reject the access and the whole query is rejected
9.              ELSE IF this is the top-level node
10.                     THEN reject the access and the whole query is rejected
11.                     ELSE send it to higher node and continue at line 1
```

Fig. 5.6. Modified Query Algorithm for the Enhanced SSM Model

5.3 Performance Evaluation

A simulator was developed to study the feasibility and efficiency of the proposed authorization model within the scope of the SSM. Performance metrics such as query response time are used to compare and contrast the enhanced SSM model against the original baseline SSM which imposes no security restriction.

First we present how an SSM hierarchy is generated for use in our simulation. Next, we describe workload and the simulation model. Finally, we examine experiments used to evaluate the performance of the two models, and discuss the results of each experiment.

5.3.1 Generating SSM Hierarchy

For the purpose of simulation, an SSM hierarchy is generated statically. The SSM hierarchy is generated based on the number of local nodes (`lnode`) and the maximum number of levels in the hierarchy (`mlevel`). The maximum number of children per SSM node (`nchild`) is computed as `lnode / mlevel + 1`. The number of children per SSM node is randomly generated between two and `nchild`. However, if there is only one node left at the lower level, the number of children per SSM node is set to one.

We assign the lowest level of the hierarchy to be *level 0* which represents local nodes. The SSM nodes start at *level 1*. The higher levels of the SSM nodes are built from their underlying SSM lower nodes until nodes at the top level are formed. Thus, higher levels have fewer nodes than lower levels.

For authorized SSM, a number of roles are generated. Each role is given a rank defined in a role hierarchy as a partial order. A rank is simply indicated by a number where zero is the highest rank with the most authority. Initially, each local node is randomly assigned a rank. Ranks are assigned to the SSM nodes according to the algorithm described in Section 5.2.2. As a result, each SSM node may contain the minimum rank or a number of ranks from lower nodes.

Since access terms at higher level nodes are more general than their children at lower level nodes, an imprecise query has higher probability to match with access terms at higher level nodes than with those at lower level nodes. The idea of using Semantic Distance Metric (SDM) in searching for a precise match can be emulated by assigning higher probability of successful match for higher level nodes than that for lower level nodes. Thus, the probability of 1.0 is equally divided and accumulated across levels. Each level is then assigned a specific *level probability*. For example, if the hierarchy has four levels, *level probabilities* ranged from the lowest level to the highest level are 0.25, 0.5, 0.75, and 1.0, respectively. We assume that all nodes at the same level have the same level probability.

5.3.2 Processing a query

When a query is submitted at a node, a randomly generated matching probability is assigned to the query. At each origin node, matching probability of the query is compared with level probability of the node. A matching probability less than or equal to the level probability represents a successful match; otherwise, it is an unsuccessful match. A successful match at a level sends the query to the lower level node(s) and an

unsuccessful match sends the query to a higher level node in the SSM hierarchy. An unsuccessful match at the top level of the hierarchy rejects the query. The query is accepted only when it is matched at local nodes.

For authorized SSM, a randomly generated rank is also assigned to a query so that it can be used to compare against ranks at each node in the query's searching path. However, the query's rank is examined only after a match is found at a node. If the query has lower rank than the node's rank, the query is rejected immediately at that node. The query is *invalid* if it is matched, but has insufficient authorization for accessing the requesting term. Hence, the output of a query can be either *accepted* at local nodes, *rejected* at the top level, or *invalid* due to insufficient authority.

5.3.3 Workload

The workload to the system is composed of precise queries and imprecise queries submitted to any node in the SSM hierarchy. Precise queries are solely executed at local nodes. Imprecise queries are moved up or down in the hierarchy until they are either accepted, rejected, or invalidated. At each node, there are a fixed number of processes (*nprocs*) running concurrently. Each process generates its own queries. When a query is resolved, a new query is generated immediately. To simplify our measurement, we assume that a query contains only one access term so that the effect of query size to the performance would be eliminated.

System parameters related to a generated SSM hierarchy are the number of local nodes, the total number of levels including the local level, the maximum number of children per SSM node, and *level probabilities* assigned to all levels. We assume that each

node takes an average of *proc_time* for processing one query. Each node also maintains a queue of queries submitted from its child nodes or its parent node. Moreover, it takes the average of *comm_time* for sending a query between any pair of nodes in the hierarchy. We assume that there is one communication link connecting a parent-child pair. All parameters and their default values are summarized in Table 5.1.

Table 5.1. Workload and System Parameters

Parameters	Default values	Description
lnode	15	number of local databases
mlevel	5	total number of levels
nchild	3	maximum number of children per SSM node
nprocs	1-4	number of processes per node
level_prob	0.2-1.0	level probability of each level
invalid_prob	0.1	probability that a query is invalid
rank_level	6	number of ranks in role hierarchy
proc_time	0.001	processing time of a node per query
comm_time	0.003	communication time between nodes

5.3.4 Performance Measurement

The performance of the enhanced SSM model (i.e., authorized SSM) is measured and compared against the original SSM model in terms of the response time and the throughput of accepted queries. Since the throughput and the response time are reciprocal, only the response time is reported. In addition, each response time reported is the average of 30 simulation runs where each run takes 3000 time units.

The response time of an imprecise query depends on several factors: i) the structure of the SSM hierarchy - Even though the algorithm presented in section 5.3.1 can generate SSM hierarchy with different height and structure, the same SSM hierarchy organization is used for all experiments reported in this chapter. ii) the location and the level of the originating node within the SSM hierarchy - Based on our experiences, the response time of queries originated at the same level of the SSM hierarchy also varies slightly. However, for the sake of simplicity, we assume that this effect is insignificant and the response time of all nodes in the same level is approximately the same. Hence, the average query response time of individual level, not node, is reported in this chapter. iii) the workload of each summary schema node is the sum of queries at that node and queries transferred from its parent and/or children nodes. As a result, the query response time is affected by the workload of each node, and consequently, on the frequency of imprecise queries generated at that node and the network traffic.

In the enhanced SSM model, imprecise queries with insufficient authority may be rejected as early as possible. This reduces the workload of each node and the network traffic in the SSM hierarchy. Consequently, the response time of accepted queries is decreased. Finally, we assume that the time taken for authorizing a query at each node is negligible.

5.3.5 Experiments and Results

We report on two experiments to show the effect of having authorization control in the SSM hierarchy under various conditions. For both experiments, precise queries

are submitted and executed only at local nodes. Additionally, the results reported are the average response time of accepted queries.

5.3.5.1 Experiment 1

This experiment measures the query response time of the same set of imprecise queries submitted to different SSM levels. The experiment is conducted for both the enhanced SSM model and the original SSM model.

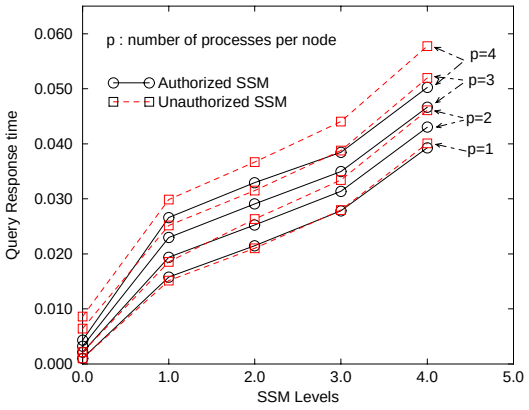

Fig. 5.7. Response time of Single SSM levels

As anticipated, in spite of the overhead due to the authorization control, the response time of accepted queries for the original SSM is mostly higher than that of the enhanced SSM (Figure 5.7). The main reason is that invalid queries in the original SSM model are travelling longer distance in the SSM hierarchy before being rejected,

whereas invalid queries in the enhanced SSM model are detected and rejected earlier. As a result, both the workload at each SSM node and the network traffic in the hierarchy of the enhanced SSM model are lower than those of the original SSM model, and hence a lower response time results.

Figure 5.7 also shows that increasing number of queries submitted simultaneously per node (*nprocs*) results in higher response time as the workload at each SSM node is increased. In the enhanced SSM model, there are two factors contributing to query response time: i) the distance between the originating node and local nodes for accepted queries, and ii) invalid queries submitted at lower levels are rejected earlier than the same queries submitted at higher levels since accessing SSM nodes at lower levels requires higher privileges of subjects.

5.3.5.2 Experiment 2

This experiment investigates the effect of the percentage of *invalid* queries on the query response time at each level. We define *invalid probability* as the ratio of invalid queries to the total number of queries submitted at all nodes of a particular SSM level. Hence, we run this experiment only for the enhanced SSM model. Moreover, for demonstration purposes, the experiments are run only for SSM level 1 and level 2.

The results are shown in Figure 5.8 and Figure 5.9, respectively. Both figures illustrate that increasing invalid probability reduces the query response time since invalid queries are detected and rejected from the SSM hierarchy earlier. The results also

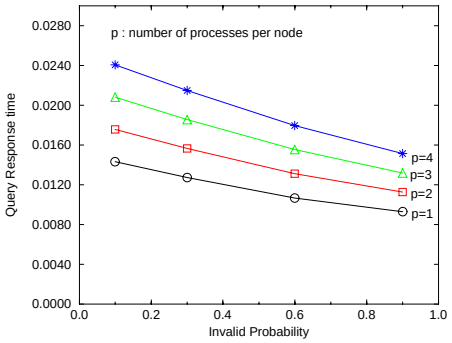

Fig. 5.8. Response time of First SSM level

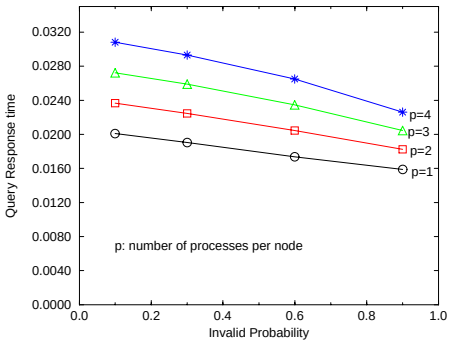

Fig. 5.9. Response time of Second SSM level

illustrate the same trend regardless of the number of queries submitted per node. Eventually, we can conclude that nodes at higher SSM levels are less sensitive to the invalid probability factor.

5.4 Related Work

The first work that addressed security issues in heterogeneous databases was presented in [49]. The authors addressed the problem of authorization inconsistency incurred when a user granted some privileges to another user at the global level, but the granting was not realized at participating local databases. As a result, a query submitted by a grantee to access data at local databases was denied. The authors modified the protection mechanism normally used in centralized databases to realize global grantings at local databases.

Most of the research in this area is directed toward the application of the security policy in traditional distributed databases and federated databases, while our work is directed toward the multidatabases (heterogeneous databases) within the scope of the SSM.

In federated databases, import/export schemas at each node classifies objects and subjects into local and global levels where authorizations are specified at both levels. Local authorizations are specified at individual local database. There are three general approaches to derive global authorizations [17]. The first approach considers that global and local authorizations can be independently specified [17]. It assumes that global objects are created and owned by the federation at the global level. Thus, global authorizations must be specified when accessing those global objects. If objects

are imported from local databases, they are not replicated at the global level and global accesses to those objects must be authorized again at local databases. Moreover, global subjects must be authenticated at the global level. To avoid inconsistent authorizations between levels, security administrators of both levels must incorporate with each other.

The second approach [26] proposes a top-down derivation that global authorizations are propagated to local databases when accesses to local objects are necessary. However, local databases may choose to accept or reject any global request. Hence, global requests may not be granted. The main problem is that global subjects defined as groups are known only at the global level. They may not be mapped to local subjects or may be mapped to many authorizations at local databases.

The third approach [13] presents a bottom-up derivation that global authorizations are derived from local authorizations for integrated and imported objects using word similarity. In addition, a dictionary is used to maintain mappings between local to global entities. Consequently, global authorizations are derived based on how close two local subjects are. The closeness is calculated based on the similarity between access compatibility of objects and access permissions of local subjects to objects. As a result, if two local subjects are not compatible or have conflict accesses to the same object, no global authorization can be derived.

Our model uses a bottom-up approach, but local subjects are mapped to a global role and global authorizations are derived according to the roles defined in a role hierarchy. The mapping can be done independently and autonomously among local databases. As a result, authorization autonomy is preserved.

5.5 Chapter Conclusion

Similar to any other databases, enforcing security in multidatabases is an important issue. However, local autonomy and heterogeneity in multidatabases make this issue relatively more difficult to enforce since local databases may have diverse sets of users and may contain objects of varying degrees of sensitivity. Deriving global authorizations by integrating underlying local authorizations may be unobtainable since subjects and objects at each local database may be incompatible. In addition, local authorizations may conflict and could not be combined to form common global authorizations.

This chapter proposed an authorization model for Summary Schemas Model (SSM) which resolves imprecise queries based on the semantic contents of a request. Hypernym, hyponym, and synonym relationships among access terms exported from local databases are the main components of the SSM which form a hierarchical metadata structure. The proposed model derives global authorizations for the SSM from authorizations of local databases. The model defines global roles and a role hierarchy indicating how global roles are related to local subjects. When an access term is exported to the SSM, its mapped authorized role is also tagged to the term. Global authorizations are then automatically derived for metadata in the SSM hierarchy based on role definitions in the role hierarchy. Since local subjects and global roles can be modified independently and autonomously, authorization autonomy is preserved in our model.

Imposing authorization information to the SSM adds both space and time overhead and clearly affects the query resolution of the SSM. The chapter evaluated the performance of the proposed model and compared it with that of the original SSM. The

simulation results showed that the proposed model outperforms the original SSM model since user queries with insufficient authority are rejected earlier. The early rejection of unauthorized queries reduces the network traffic and workload at both SSM nodes and at local databases. Thus, the response time of valid queries in the proposed model is lower than that of the original SSM.

Chapter 6

Concluding Remarks and Future Work

With recent growth of interest in global information sharing due to both communication and computation technology, security as well as performance issues have gained much attention. In this thesis, we investigated the impact of security requirements in selected database platforms on the design choices and the performance of applications running on them. We also proposed designs that not only meet security requirements, but also achieve good performance.

6.1 Contributions of this Thesis

This thesis focused on three selected database platforms: multilevel databases, replicated databases and multidatabases because each system has its own unique security requirements. For each database platform, security requirements were identified, a secure solution for an application was proposed and its performance was evaluated. The contributions of our work are summarized below.

First, in multilevel databases, preventing potential covert channels is a major concern. We designed and implemented a multilevel log manager running under a multilevel security policy. The proposed design avoids three well-known covert channels in a multilevel log manager. The design is implemented as part of the experimental multilevel secure database system, STAR-DBS, running under DTOS. DTOS is a secure operating

system which supports a multilevel security policy. Security features provided by DTOS help separate the operation of each security level so that the noninterference property can be ensured. In addition, the multilevel log manager must execute in a trusted domain for serving requests from every security level. As illustrated in the experiments, our multilevel log manager is simpler and offers better performance than that of [39].

Second, in replicated databases, maintaining the consistency of a common security policy replicated among several machines is required to prevent security violations. We proposed invalidation-based consistency protocols that ensure one-copy serializability and do not allow transient inconsistency. The simulation results indicate that the proposed protocols outperform update-based consistency protocols.

Third, in multidatabases, preventing unauthorized accesses to metadata in the Summary Schemas Model (SSM) hierarchy is mandatory since the majority of of unauthorized accesses is rejected from the system before arriving at local databases, and hence authorized accesses can arrive at local databases faster. As a result, the vulnerability of the SSM hierarchy to denial-of-service attacks is diminished. We proposed an authorization model for the SSM based on a role-based security policy since its role hierarchy fits naturally with the hierarchical structure of the SSM. With respect to authorization autonomy of local databases, our global authorizations can be derived from individual local authorizations using global roles defined by the MDBS. In the proposed model, unauthorized accesses are detected earlier, and hence network traffic and computation load in the SSM hierarchy are lessened. The simulation results show that the proposed model gives better performance than the original SSM model. As a result, both the security and the overall performance of the SSM are enhanced.

In summary, we have presented three topics to illustrate that, even under security requirements of each database system, the applications are able to achieve good performance. We believe that our ideas presented here will lead to the design of secure and efficient applications.

6.2 Future Research

The proposed work in this thesis can be extended in several directions. A possible list is presented below.

- During the design of our multilevel log manager, we prevent disk scheduling covert channel by having logs of each security level written to separate disk. However, the design may lead to underutilization of disk space for high-level activities as low-level activities are typically higher than high-level activities. In addition, the design is not generalized since accessing shared data across security levels is not feasible. One solution is to design multilevel disk scheduling algorithms that would prevent the completion of low-level disk requests from being interfered by the completion of high-level disk requests. Algorithms for channel-free disk scheduling presented in [27, 51] mask the completion order and the completion time of disk requests of one level from being interfered by those of another level. Intuitively, the algorithms increase the average seek time of all security levels. However, their performance has not been extensively investigated. Moreover, other possible and better solutions have not been explored.

- Invalidation-based consistency protocols presented in this thesis assume that the primary copy of the security policy database is maintained only at the security server. This may become a bottleneck since every object manager has to request a new copy of the policy whenever there is a change in the policy. To alleviate this bottleneck, we may partition the policy and store it at several sites. Each site holds the primary copy of a partition. Changing a portion of the policy should be allowed at its primary site only. This avoids inconsistency due to simultaneous multiple updates of the same replica and hence no reconcilation is necessary. In addition, we may extend the protocols to include both update and invalidation.

- In our performance study of the enhanced Summary Schemas Model, user queries are submitted to SSM nodes of one level at a time. This does not reflect a practical situation where users may submit their queries at any node and at any time. In addition, we simply assume that each query contains only one imprecise term. Hence, an extensive performance study should be carried out for simulated real workload, and also for various SSM hierarchical structures.

- Update issues in multidatabases are complicated due to local autonomy and heterogeneity [10]. Most studies focus on concurrency control and consistency [28, 29] where most commonly used mechanisms such as two-phase locking have to be relaxed for performance gain. However, maintaining consistency of security policy in multidatabases has not been addressed. In the enhanced Summary Schemas Model proposed in this thesis, global authorizations can be derived without violating local authorizations. However, local databases are likely dynamic and changes

in security policy may occur more often than those in a single database. Thus, global authorizations would be certainly affected by such changes. It would be beneficial to study (i) consistency protocols for maintaining security policy under the enhanced SSM model; and (ii) how dynamic changes in local and global authorization levels have an impact to the overall performance of the SSM model.

- Summary Schemas Model was extended to include mobility and wireless communication [28]. A mobile computing environment introduces extra requirements such as disconnectivity and limited bandwidth to the model. Even though the issues of its concurrency control, recovery and replication have been studied [28], security issues in mobile environment under the scope of the SSM platform have not been addressed. The authorization model proposed for Summary Schemas Model in this thesis can be extended to include mobile data accesses.

References

[1] M. Accetta, W. Bolosky, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, GA, June 1986.

[2] E. Amoroso. *Fundamentals of Computer Security Technology.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[3] T. Anderson, Y. Breitbart, H.F. Korth, and A. Wool. Replication, Consistency, and Practically: Are These Mutually Exclusive? In *Proceedings of ACM SIGMOD Conference*, pages 484–495, Seattle, WA, May 1998.

[4] A. Baskaran. A Slot Stealing Based Multilevel Secure Database Buffer Manager. Master's thesis, Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, December 1996.

[5] D. Bell and L. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretations. Technical Report MTR-2997, Mitre Corporation, March 1976.

[6] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[7] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann, San Francisco, CA, 1997.

[8] W.E. Boebert and R.Y. Kain. A Practical Alternative to Hierachical Integrity Policies. In *Proceedings of 8th National Computer Security Conference*, pages 18–27, Gaithersburg, MD, October 1985.

[9] Y. Breitbart, R. Komondoor, R. Rastogi, and S. Seshadri. Update Propagation Protocols for Replicated Databases. In *Proceedings of ACM SIGMOD Conference*, pages 97–108, Philadelphia, PA, June 1999.

[10] Y. Breitbart and A. Silberschatz. Multidatabase Update Issues. In *Proceedings of the Conference of Management of Data*, pages 135–142, 1988.

[11] M. W. Bright, Ali R. Hurson, and S. Paksad. Automated Resolution of Semantic Heterogeneity Multidatabases. *ACM Transactions on Database Systems*, 19(2):212–253, June 1994.

[12] O.A. Bukhres and A.K. Elmagarmid. *Object-Oriented Multidatabase Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1996.

[13] S. Castano. An Approach to Deriving Global Authorizations in Federated Database Systems. In *Proceedings of the IFIP Working Conference on Database Security*, pages 58–75, Como, Italy, July 1996.

[14] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security.* Addison-Wesley, 1995.

[15] P. Chundi, D.J. Rosenkratz, and S.S. Ravi. Deferred Updates and Data Placement in Distributed Databases. In *Proceedings of the 12th International Conference on Data Engineering*, New Orleans, LA, 1996.

[16] D.E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[17] S.D.C. di Vimercati and P. Samarati. An Authorization Model for Federated Systems. In *Proceedings of 4th European Symposium on Research in Computer Security*, pages 99–117, Rome, Italy, September 1996.

[18] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, Reading, MA, 3 edition, 2000.

[19] M.J. Franklin, M.J. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database Systems*, 22(3):315–363, September 1997.

[20] D. Gawlick, J. Gray, W. Limura, and R. Obermarck. *Method and Apparatus for Logging Journal Data Using a Log Write Ahead Data Set*. U.S.Patent 4507751 issued to IBM, March 1985.

[21] J. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982.

[22] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–97, San Francisco, CA, June 1990.

[23] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, 1993.

[24] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of ACM SIGMOD Conference*, pages 173–182, Montreal, Quebec, Canada, May 1996.

[25] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, 1976.

[26] D. Jonscher and K.R. Dittrich. An Approach for Building Secure Database Federations. In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994.

[27] P.A. Karger and J.C. Wray. Storage Channels in Disk Arm Optimization. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 52–61, Oakland, CA, May 1991.

[28] J.B. Lim and A.R. Hurson. Transaction Processing in a Mobile, Multi-Database Environment. *Multimedia Tools and Applications*, 15:161–185, 2001.

[29] S. Mehrotra, H.F. Korth, and A. Silberschatz. Concurrency Control in Hierarchical Multidatabase Systems. *The VLDB Journal*, 6(2):152–172, 1997.

[30] S.E. Minear. Providing Policy Control over Object Operations in a Mach Based System. In *5th USENIX Unix Security Symposium*, pages 1–15, Salt Lake City, UT, April 1995.

[31] J. D. Moffett and E. C. Lupu. The Uses of Role Hierarchies in Access Control. In *Proceedings of 4th ACM Workshop on Role-Based Access Control*, pages 153–160, Fairfax, VA, October 1999.

[32] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[33] T. Morzy and Z. Krolikowski. Query Optimization in Multidatabase Systems: Solutions and Open Issues. In *Proceedings of 10th International Conference in Database and Expert Systems Applications*, pages 6–11, September 1999.

[34] S. Ngamsuriyaroj, T.F. Keefe, and A.R. Hurson. Maintaining Consistency of the Security Policies in Distributed Environment. To appear in IPCCC 2002, April 2002.

[35] S. Ngamsuriyaroj, T.F. Keefe, and A.R. Hurson. Maintaining Consistency of the Security Policies using Timestamp Ordering. To appear in ITCC 2002, April 2002.

[36] S. Osborn. Database Security Integration using Role-Based Access Control. In *Proceedings of the IFIP Working Conference on Database Security*, pages 1–15, Schoorl/Amsterdam, Netherlands, August 2000.

[37] E. Pacitti and E. Simon. Update Propagation Strategies to Improve Freshness in Lazy Master Replicated Databases. *The VLDB Journal*, 19(8):305–318, 2000.

[38] C.S. Park, M.H. Kim, and Y.J. Lee. A Replica Control Method for Improving Availability for Read-Only Transactions. In *Proceedings of International Database Engineering and Applications Symposium*, pages 104–112, August 1997.

[39] V.R. Pesati, T.F. Keefe, and S. Pal. The Design and Implementation of a Multilevel Secure Log Manager. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 55–64, Oakland, CA, May 1997.

[40] C.P. Pfleeger. *Security in Computing*. Prentice-Hall, Englewood Cliffs, NJ, 2 edition, 1997.

[41] P. Samarati, P. Ammann, and S. Jajodia. Maintaining Replicated Authorizations in Distributed Database Systems. *Data and Knowledge Engineering*, 18(1):55–84, 1996.

[42] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based Access Control Models. *IEEE Computer*, 19(5):38–47, February 1996.

[43] Secure Computing Corporation. Assurance in Fluke Microkernel Formal Top Level Specification. Technical Report DTOS CDRL A004, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113, February 1999.

[44] A.P. Sheth and J.A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[45] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *The Proceedings of 8th USENIX Security Symposium*, pages 123–139, Washington, D.C., August 1999.

[46] R. Sripada and T.F. Keefe. Version Management in the STAR MLS Database System. In *Proceedings of the IFIP Working Conference on Database Security*, pages 159–171, Chalkidiki, Greece, July 1998.

[47] A.S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[48] Z. Tari and G. Fernandez. Security Enforcement in the DOK Federated Database System. In *Proceedings of the IFIP Working Conference on Database Security*, pages 23–42, Como, Italy, July 1996.

[49] C.Y. Wang and D. Spooner. Access control in a heterogeneous distributed database management system. In *Proceedings of the 6th Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg, VA, March 1987.

[50] A.C. Warner and T.F. Keefe. Version Pool Management in a Multilevel Secure Multiversion Transaction Manager. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 169–182, Oakland, CA, May 1995.

[51] J.C. Wray. An Analysis of Covert Timing Channels. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 2–7, Oakland, CA, May 1991.

Vita

Sudsanguan Ngamsuriyaroj received both BS and MS from Mahidol University, Thailand. In Fall 1995, she enrolled in the Department of Computer Science and Engineering at Penn State University to pursue her Ph.D. degree. She worked as a teaching assistant for the department from Spring 1996 to Spring 1999. Research areas of interest include database security, computer and network security, transaction processing and performance evaluation.