

**SUPPORTING SECURE AND EFFICIENT
WRITE-UP IN HIGH-ASSURANCE MULTILEVEL
OBJECT-BASED COMPUTING**

by

Roshan K. Thomas

A Dissertation

Submitted to the

Graduate Faculty

of

George Mason University

in Partial Fulfillment of

the Requirements for the Degree

of

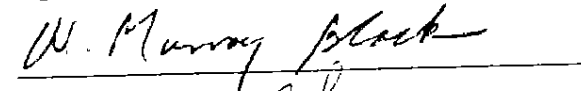
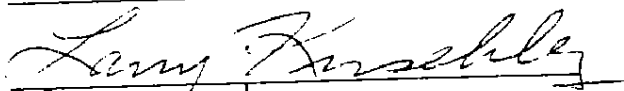
Doctor of Philosophy

Information Technology

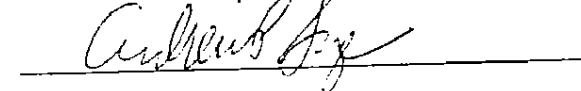
Committee:



Director



Department Chairperson



Dean of the School of Information
Technology and Engineering

Date: 8/05/94

Summer 1994

George Mason University

Fairfax, Virginia

Acknowledgments

A WORD OF THANKS AND A PRAYER

To many I owe a debt beyond measure,
and I know not how to repay;
in each one I have found a treasure,
and the heavens recompense them. I pray.

Countless were the toils and moments of despair.
but never did I doubt that grace was abound:
many a day I searched my faculties, only to find them bare,
but I knew someday I would be homeward bound.

As I bid farewell to these shores and sail to the real world,
to work the looms of life and practice my trade:
may I be quick to praise, and slow to scold.
and patiently heedful of life's many shades.

As I receive this short ovation,
may I learn to grow in Purity, humility, and Devotion.

R.K.T

I wish to express my deepest gratitude to my advisor Ravi Sandhu for his guidance, support, and encouragement in enabling me bring this dissertation to completion. I have benefited greatly from his foresight, attention to detail, and thoroughness. His overall guidance has had a profound effect on this work, and he will continue to be an influencing force and model for the rest of my life.

I also wish to thank the members of my committee Paul Ammann, Larry Kerschberg, and David Rine. I thank Dr. John Mclean of the Naval Research Laboratories for an afternoon of valuable suggestions. I also thank Sushil Jajodia for supporting me during the initial stages of my study.

Over the course of the last five years of this doctoral journey, I have made several friends in Washington who have helped me in my personhood, and in doing so, contributed to this work. I thank Dick and Audrey Webb for being my host family in Washington. You have truly made me feel as a member of your family, and my love for all of you will remain forever. I am truly grateful to the gang: Sunil Chacko, Suresh and Rosemarie Varghese, Margaret Mathews, Gwen DeMiranda, Roger Vales, and Luis Sanchez. Washington would not have been the same without their friendship. I am indebted to Brian and Wendy Samuels for their friendship and love over the years. I also thank Doreen and John John for their friendship and the influence they have had in my life. Their inexhaustible supply of love always came at the right time. I also thank my office mates and friends Vijayalakshmi Atluri, Srinivas Ganta, Indrakshi Mukherji, and Indrajit Ray. They have provided a stimulating work environment.

Last but not the least, I am deeply grateful to my parents Thomas and Valsa, my brothers Riju, Rolin, Roy, and sister Rinta, for their love and support, and prayers. Over the years, they have nurtured me from across the miles, from three continents. They have, and always will be, my anchor and inspiration. I dedicate this dissertation to them.

Table of Contents

	Page
List of Tables	vii
List of Figures	viii
Abstract	x
Chapter 1. Introduction	1
1.1 Object-Based Computing	3
1.1.1 Concurrency and Message-Passing Alternatives	4
1.1.2 Persistent and Sharable Objects	5
1.2 Multilevel Security	6
1.2.1 Lattice-based Security Models	6
1.2.2 Covert and Signaling Channels	10
1.2.3 Security Kernels, Architectures, and Trusted Subjects	12
1.2.4 Multilevel System Architectures	13
1.3 Multilevel Security in Object-based Computing	16
1.4 Summary of Previous Work	20
1.5 Organization of Thesis	21
 Chapter 2. Motivation and Problem Statement	 23
2.1 Motivating Applications and Examples	25
2.1.1 Payroll Processing Example	25
2.1.2 Situation Assessment Example	27
2.2 Write-Up and Confidentiality, Integrity, and Performance Tradeoffs	28
 Chapter 3. Message Filtering and Mandatory Security Enforcement	 34
3.1 The Message Filter Model	34
3.1.1 The Message Filtering Algorithm	35
3.1.2 Restricted Methods and Invocation Trees	37
3.2 Implementing Message Filtering	39

3.2.1	System Layering and the Security Perimeter	39
3.2.2	An Executable Specification	42
Chapter 4.	Asynchronous Computing with RPC Semantics	46
4.1	Serial Correctness versus Concurrency	46
4.2	Maintaining Global Serial Fork Order	51
4.3	A Family of Scheduling Strategies	55
4.3.1	A Conservative Level-by-level Scheduling Scheme	55
4.3.2	An Aggressive Scheduling Scheme	58
4.3.3	Hybrid Schemes	60
4.3.4	A Framework and Metric for Comparative Analysis	60
Chapter 5.	Trusted Subject Architecture Implementation	69
5.1	Architecture	69
5.2	Scheduling Algorithms	70
5.3	Proof of Confidentiality	80
Chapter 6.	Kernelized Architecture Implementation	89
6.1	Architecture	89
6.2	Scheduling Algorithms	90
6.2.1	Conservative Scheduling Algorithms	91
6.2.2	Aggressive Scheduling Algorithms	98
Chapter 7.	Replicated Architecture Implementation	104
7.1	Architecture	104
7.2	Message-filtering in the Replicated Architecture	107
7.3	Serial Correctness and Replica Control	109
7.4	Scheduling Algorithms	115
7.4.1	Conservative Scheduling Algorithms	116
7.4.2	Aggressive Scheduling Algorithms	120
Chapter 8.	Inter-session Synchronization	131
8.1	Inter-session Concurrency for the Kernelized Architecture	131
8.1.1	Multilevel Checkin/Checkout of Objects	132
8.1.2	Checkin/Checkout Variations	134
8.2	Inter-session Concurrency for the Trusted Subject Architecture	137
8.3	Inter-session Concurrency for the Replicated Architecture	138

Chapter 9. Summary and Conclusions	145
9.1 Research Contributions	145
9.2 Future Work	147
Bibliography	150

List of Tables

Table	Page
1.1 Summary of main results	22

List of Figures

Figure	Page
1.1 An example of Trojan horse leaking information	9
1.2 A kernelized multilevel system architecture	14
1.3 A multilevel system architecture with trusted subjects	14
1.4 The replicated multilevel system architecture	15
1.5 Object classification granularities	18
2.1 Objects in a payroll database	25
2.2 Write-up in situation assesement	29
2.3 A writeup message and its reply	30
3.1 Message filtering algorithm	36
3.2 Restricted methods in a chain	37
3.3 A tree with restricted subtrees	38
3.4 A layered architecture with TCB	40
3.5 Message manager algorithms for SEND and QUIT	45
4.1 A tree of concurrent computations	48
4.2 Generation of forkstamps for a session's computation tree	52
4.3 Conservative level-by-level scheuling in lattice and partial order	56
4.4 Progressive execution under conservative scheduling	65
4.5 Progressive execution under aggressive scheduling	66
4.6 Progressive execution under hybrid scheduling	67
4.7 Full-enablers for a longest maximal chain of 3 elements ($n = 3$)	68
4.8 Full-enablers for a longest maximal chain of 4 elements ($n = 4$)	68
5.1 A trusted subject architecture with TCB	71
5.2 Session manager algorithm for FORK	73
5.3 Session manager algorithm for START	74
5.4 Session manager algorithm for TERMINATE	75
5.5 The two-step processing cycle of the session manager	81

5.6	Illustrating the noninterference proof	S5
6.1	A kernelized architecture	90
6.2	Level manager algorithm for terminate processing	91
6.3	Level manager algorithm for fork processing	92
6.4	Level manager algorithm for wake-up processing	92
6.5	Level manager algorithm for start processing	93
6.6	Processing fork requests under aggressive scheduling	99
6.7	Processing wake-up requests under aggressive scheduling	99
6.8	Processing terminate requests under aggressive scheduling	100
7.1	The replicated architecture illustrating containers for a simple lattice	105
7.2	A transaction tree and its subtransaction mapping	111
7.3	Processing fork requests under conservative scheduling	120
7.4	Processing terminate requests under conservative scheduling	121
7.5	Processing wake-up requests under conservative scheduling	122
7.6	Processing posted updates with conservative scheduling	122
7.7	Updating the local container before starting a transaction	123
7.8	Processing fork requests under aggressive scheduling	125
7.9	Processing wake-up requests under aggressive scheduling	126
7.10	Processing terminate requests under aggressive scheduling	127
7.11	Processing posted updates	128
7.12	Recording the fork of computations at lower levels	128
8.1	Illustrating histories generated with various inter-session synchronization schemes	140

Abstract

SUPPORTING SECURE AND EFFICIENT WRITE-UP IN HIGH-ASSURANCE MULTILEVEL OBJECT-BASED COMPUTING

Roshan K. Thomas, Ph.D.

George Mason University, 1994

Dissertation Director: Dr. Ravi Sandhu

This dissertation addresses the support of secure and efficient remote procedure call (RPC) based synchronous write-up actions in multilevel object-based computing environments and systems. These environments and systems are characterized by objects classified at varying security levels (called classifications) and accessed by subjects with varying security clearances. Many multilevel systems, such as relational multilevel database management systems, typically do not allow write-up, due to integrity problems arising from the blind nature of write-up operations in these systems. (A blind write-up operation is one that is not allowed to read the data at a higher level, but is allowed to overwrite it.) In object-based computing environments, sending messages upwards in the security lattice does not present an integrity problem because such messages will be processed by appropriate methods in the destination (receiver) object. However, supporting write-up operations in object-based systems is complicated by the fact that such operations are no longer primitive; but can be arbitrarily complex and therefore can take arbitrary amounts of processing time. Dealing with the timing of write-up operations consequently has broad implications on confidentiality (due to the possibility of signaling channels), integrity, and performance.

We present an asynchronous computation model for multilevel object-based computing, which achieves the conflicting goals of confidentiality, integrity, and efficiency. This requires concurrent computations (methods) to be generated whenever write-up actions are issued, and for them to be scheduled and synchronized so that the net effect is logically that of a sequential computation (mimicing remote procedure call semantics). The computations generated by a user form a tree of concurrent computations and are collectively considered to belong to a user session. Our work utilizes an underlying message filter security model to enforce mandatory confidentiality. We demonstrate how our computation model can be implemented within the framework of trusted subject, kernelized, and replicated architectures. In doing so, we present a family of synchronization and scheduling schemes for concurrent computations, and present a framework for comparative analysis. We also present various inter-session synchronization schemes.

Chapter 1

Introduction

The object-oriented paradigm continues to emerge as a useful and unifying one in computer science and engineering. Ideas from the paradigm have been incorporated in such diverse fields as software engineering, artificial intelligence, and database management, with the resulting advancement of these fields in new directions. The interest of the artificial intelligence community to object-orientation has come from the search for knowledge representation and management schemes for large knowledge bases. From the software engineering perspective, objects are seen as instances of abstract data types (ADT's) [LZ74] incorporating the principles of encapsulation and information hiding. It is now well known that these are vital in building software systems and components that are highly modular, maintainable, and reusable. From the database viewpoint, the object-oriented data model overcomes the limitations of record-based models by being able to model complex structures, relationships, and behavior. These capabilities are essential for emerging application areas such as computer-aided design and manufacturing (CAD/CAM), software management and reuse, and office information systems, to name a few.

Central to the object-oriented paradigm is the notion of an *object*. An object models some real-world entity, encapsulates some private state, and is identified by some persistent and unique identity. An object also encapsulates services which are made available to requesting clients through a public interface. Services are

implemented by pieces of code called methods, and invoked by sending messages to objects.

Although there is wide agreement on the above primitive notion of objects, much debate has ensued in the last few years over the formulation of an *object-oriented data model*. There is some consensus on the core concepts that such a data model should support. However, a single universal definition has not emerged [Mai89]. Some researchers also distinguish between object-based, class-based, and object-oriented systems. In Wegner's view [Weg87], a language is considered to be *object-based* if it provides linguistic support for objects. Examples of such languages include Actors [Agh87], and Ada. In addition, if a language supports the notion of classes, it is considered to be *class-based*. An example of such a language is Clu [L+77]. Further, if the language supports the notion of inheritance, it is categorized as being *object-oriented*. Inheritance allows objects and classes to share structure in terms of inherited attributes and behavior in terms of inherited methods.

In this dissertation, we address the support of object-based computing in multilevel-secure environments. Such environments are characterized by objects labeled at different classifications, and accessed by users (or more precisely subjects on their behalf) cleared to various security levels. A security policy governs how the subjects can access the various objects in the system. We are not concerned with the plethora of modeling issues and variations in the object-oriented data model. Rather, we are interested in how objects can form a basic model for secure computing. Computing in the object framework reduces to sequences of message passing and method invocations among objects. We are thus interested in how objects can send messages and exchange information without violating the relevant security policy.

It is now widely recognized that computer and information security consists of three distinct but interrelated areas, namely *confidentiality*, *integrity* and *availability*.

Confidentiality is concerned with the disclosure of information, integrity is concerned with the improper modification of information, and availability is concerned with the denial of access to information. In this dissertation our focus is primarily on confidentiality and integrity issues.

Before proceeding further in the discussion, it is helpful to clarify the distinction between *users*, *principals*, and *subjects*. By a user, we mean a human being, represented in the system by a unique user identity. Each user may be associated with several principals. However, a principal can be associated with only a single user. Each principal associated with a single user may be given a different set of access rights, and allowed to login at different security levels as long as these levels are dominated by the clearance of the parent user. Each principal may in turn have several subjects. Each subject is a process in the system.

The rest of this introductory chapter is organized as follows. The first section discusses concurrency, message passing, and other issues in object-based computing that are relevant to the topic of this dissertation. This is followed by a brief introduction to multilevel security. The last section highlights the organization of the rest of the dissertation.

1.1 Object-Based Computing

The wide applicability of the object-oriented paradigm and way of thinking has naturally led to the integration of many technologies. For example, a noticeable trend is the enhancement of object-oriented programming languages with support for objects that are persistent and sharable. The ability to support persistent and sharable objects is indeed the main feature offered by database management systems. Thus the distinction between object-oriented programming environments and databases is increasingly becoming blurred. Another wave of integration will likely result from

concurrent object-based computing environments incorporating database functionality. Application areas such as situation assessment, network monitoring, and process control are naturally modeled as cooperating concurrent objects. However, the states of such objects may be required to persist beyond individual computing sessions and process lifetimes, and to be shared by other objects. For example, in a process control system, persistency may be required for the simple reason that we need history information for measurements of quantities such as temperature and pressure.

The issues addressed in this dissertation are most pertinent to domains which can be modeled as a set of autonomous but cooperating objects such that the object-based model of computing is a good fit, and where object states are required to be persistent and sharable. The mechanisms we propose can be incorporated into a variety of multilevel computing environments and systems such as object-oriented databases and message-based operating systems.

1.1.1 Concurrency and Message-Passing Alternatives

The object-based model of computing sees the world as a set of concurrent and cooperating objects. As such these objects would have to occasionally interact with each other and exchange information. An obvious approach would be to use common variables residing in some shared memory. However object interaction based on message passing is the most suitable, when objects are autonomous entities executing in a loosely coupled environment. We may categorize message passing alternatives into two categories:

- **Synchronous.** In the synchronous form of message passing, a sender object O_1 sends a message to a receiver object O_2 , and is suspended until the message is delivered to the intended receiver and a reply returned.

- **Asynchronous.** In this case the sender transmits a message and continues execution without waiting for the message to be delivered, or for the reply.

Synchronous message passing essentially parallels the semantics of remote procedure calls (RPC's), with the difference that the receiver's activity does not have to end with the return of the reply. The asynchronous form varies in style from the sending of a single message to more complex forms involving streams of messages. In either case, asynchronous message passing requires synchronization if the sender needs to access the reply returned from the receiver. This is because the sender and the receiver may be executing concurrently. The well-known approach to such synchronization is based on the notion of *futures* [Weg91]. A future is a data structure (object) that represents the results of the concurrently executing receiver object (process). When a message is sent in asynchronous mode, a future is created, after which the sender continues execution. The future represents a promise or I.O.U. from the receiver object (the called process). When the sender subsequently wishes to obtain the reply it accesses the future object. If the promise has not been fulfilled, the sender has to wait until the receiver returns the required results to the future object.

In the multilevel context, whenever a sender has to wait for a reply or some other result, there always exists the possibility of confidentiality leaks through covert channels (as discussed in subsection 1.2.2). This may occur from both synchronous and asynchronous communications. However, in our subsequent discussions, we focus on synchronous (RPC-based) communications as the intended correctness semantics is clear.

1.1.2 Persistent and Sharable Objects

When objects persist beyond individual user sessions and are shared by many users, the integrity of the data objects becomes a primary concern. The actions or updates

of one user on an object can causally influence the values read and written by other users. Further, information can flow from one object to another.

Several definitions of data integrity have appeared in recent literature. Sandhu has succinctly compared and summarized these different notions of integrity [San93]. Of these different notions of integrity, the one most suitable in our context is that which is concerned with the improper modification of data. Our objective is to make sure that objects are modified in a fashion that is consistent with a synchronous message passing (remote procedure-call) semantics. We are not concerned with data integrity in the sense of an expectation of data quality that may incorporate liveness requirements.

1.2 Multilevel Security

The notion of multilevel security for data confidentiality originated in the late 1960's when the U.S. Department of Defense wanted to protect classified information processed by computers. Environments and applications requiring multilevel security are characterized by users with more than one clearance level sharing data with more than one sensitivity level (classification).

In the following subsections, we review basic notions of multilevel security by introducing lattice-based security policies and models. We then discuss how even with adequate access control mechanisms, information may still leak through covert and signaling channels.

1.2.1 Lattice-based Security Models

The military security policy is a special case of a more general lattice-based security policy. Every object in the system is assigned a security class (also known as a security

label). Information is allowed to flow between two objects only if the policy allows information to flow between the corresponding classes. Given a set SC of security classes, we can formally define a binary can-flow relation $\rightarrow \subseteq SC \times SC$. It is also convenient to define the inverse of the can-flow relation called the dominates relation. We say $A \geq B$ (A dominates B) if and only if $B \rightarrow A$.

In a lattice-based approach to multilevel security, the security classes form a mathematical structure called a lattice. The security elements of the lattice are partially ordered under the can-flow (\rightarrow) relation. In a lattice, there may be pairs of elements say (A, B) for which the can-flow relation does not hold (i.e., $A \not\rightarrow B$ is true). Every pair of elements in a lattice possesses a *greatest lower bound*. In other words, for every pair of elements (A, B) there is an element L such that $L \rightarrow A$ and $L \rightarrow B$ hold. Similarly, every pair of elements in a lattice has a *least upper bound*.

In the military and government setting, the security label given to a data item (object) consists of two parts, a *hierarchical level* and a *category*. The set of hierarchical levels is totally ordered as follows:

Top-secret > Secret > Confidential > Unclassified

The individual set elements in a category are known as *compartments*. Compartments are used to implement the principle of least privilege or "need-to-know". This is a well known principle in security and ensures that a user has access to the minimum number of objects required to perform his or her job. The compartment assigned to an object typically reflects the subject matter of the information contained in the object. We say a security level dominates (\geq) another if its hierarchical level is greater than the other's and its category set includes the others. As an example consider two compartments CRYPTO and ATOMIC. Thus the level <top-secret; CRYPTO, ATOMIC> dominates the level <secret; CRYPTO>.

Having introduced lattice-based security policies, we now turn our attention

to related security models. A security model is used to implement a security policy. The Bell and LaPadula security model (also called the BLP model) was the first to formally address multilevel security, and even today remains the *de facto* standard [BL76]. BLP characterizes and governs access control and information flow with the following two rules (l denotes the label of the corresponding subject (s) or object (o)).

- **Simple Security Property.** Subject s can read object o only if $l(s) \geq l(o)$.
- **\star -Property.** Subject s can write object o only if $l(s) \leq l(o)$.

The need for the simple-security rule is obvious; it prevents low level users (and subjects) from reading information stored at higher levels. It thus prevents “read-up” operations. This requirement parallels that of the paper world with documents and human beings (users). However, it turns out that disallowing read-up operations alone is not sufficient to prevent illegal information flows that violate the security policy. To illustrate, a high subject may read a file classified at high, and write a subset of its contents (or information derived from its contents) into a second file at a lower file. This would clearly violate the security policy as information is flowing downwards in the security lattice. The \star -property (pronounced star property) prevents such violations by disallowing write-down operations.

The \star -property is really a confinement property and at first sight might appear to be overly restrictive and counter-intuitive to the way the real (paper) world works. A human user cleared to a high level may talk and disclose information to users at lower levels. However, in the real world, the high user is “trusted” to exercise judgement and not disclose any highly classified information. In other words the user is allowed to talk about public information such as sports scores and the weather. A subject, which is a program or process in the system, cannot be trusted in a similar fashion.

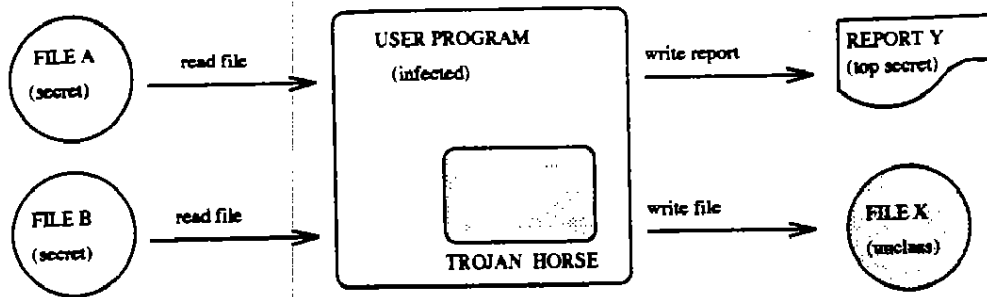


Figure 1.1: An example of Trojan horse leaking information

The above distinction between the trust placed in human users and the mistrust of subjects is significant and fundamental to implementing security policies in computer systems. A subject (computer program) cannot be trusted as it may contain bugs and trojan horses. A *trojan horse* is a malicious program which in addition to its stated objectives, performs some hidden functions. Trojan horses are typically embedded into application programs and utilities. A typical victim such as an end user, is not aware of the Trojan horse when using the infected program. Fig 1.1 illustrates how information could be leaked with the help of a Trojan horse. A high subject (program) reads two high level files and compiles a high level report. Meanwhile, the Trojan horse that is embedded in the program, writes some information in the high level files to a low level file. The \star -property prevents such leaks.

The Bell-LaPadula rules are examples of what we call *mandatory access control* (MAC) rules. Mandatory access control places restrictions on the access of objects based on their security labels, and the controls themselves cannot be bypassed. In many models and systems (including BLP), mandatory access control is complemented by *discretionary access control* (DAC) mechanisms. In the DAC framework, objects are owned by subjects who at their discretion, propagate rights to these objects, to other subjects. An example of a discretionary access control rule would be one that specifies how the owner of a directory may give read or write permission to the directory, to other subjects. In this dissertation we do not address discretionary

access control issues as they are irrelevant to the problems associated with supporting write-up actions.

1.2.2 Covert and Signaling Channels

In our discussion so far, we have seen how mandatory access controls prevent illegal communication (information flows) between access classes. However, a system that enforces mandatory access controls such as one based on the BLP model can be riddled with *covert channels* (also called leakage paths). These channels arise due to processes sharing resources in the system, and pose a formidable problem in building secure systems. Models such as BLP approach access control from a certain level of abstraction. However, there exists several shared resources and variables such as buffer pools and global counters that are not part of the abstractions of BLP, but nevertheless are shared by processes (subjects) at multiple security levels.

There exists two types of covert channels. A *storage channel* arises when a process writes an object or variable and another process can observe or read the effect of the write. A *timing channel* results when the activity of a high level subject affects the performance of the system in such a way that it can be observed and measured by lower level subjects. To exploit timing channels, subjects must have the ability to measure time (such as having access to a real-time clock). Storage channels do not require access to any such timing base.

As an illustration, let us see how a storage covert channel based on resource exhaustion can be formed. Consider a computer system that has a 10MB pool of dynamically allocated memory. A high subject requests the entire pool of memory and its request is granted. This is followed by a request from a low subject for 10MB of memory. Obviously, this request cannot be granted, and the low subject records this fact as one bit of information. Now a colluding high subject can selectively

request and release the pool of memory at regular intervals, causing the low subject's request to be denied or honored in a specific pattern, and thereby opening up the covert channel. Another example of a storage channel is one that arises from a low subject inferring the existence of an object at a higher level. A low subject may accomplish this by attempting to create a file with a file-name that already exists, and the file system rejecting the request since it has to guarantee the uniqueness of file names. A pair of colluding subjects can cause a pattern of bits to be leaked. Lastly, a simple illustration of a timing channel is one that is caused by the modulation of CPU utilization. A high subject can vary the CPU utilization at some constant interval, causing the low subject's progress to be modulated and measured.

In summary, a covert channel is a communication channel not normally intended for direct communication between subjects in the system. These channels are beyond the purview of abstract security and access control models such as BLP. Detecting and closing them will require analysis of information flow within the internals of individual systems.

In this dissertation we distinguish between the above mentioned covert channels and *signaling channels* [JS91]. A signaling channel is a means of downward information flow which is inherent in a data or computation model, and will therefore occur in *every* implementation of the model. A covert channel on the other hand is a property of a specific implementation, and not a property of the data or computation model. In other words, even if the data or computation model is free of downward signaling channels, an implementation may well contain covert channels due to implementation specific quirks. Conversely, closing all covert channels in an implementation will not eliminate signaling channels. It should be noted however that both covert and signaling channels form illegal and unintended communication paths.

As mentioned before, the object-based computation model essentially reduces to sequences of message passing among objects. Unfortunately, with such a computation model, message passing with remote procedure-call (RPC) semantics is vulnerable to signaling channel attacks, as we will see shortly. This has broad implications for building multilevel-secure systems that support object-based computing. In particular, solutions to close signaling channels should be addressed to the layer that supports the data and computation model, rather than fine-tuning low-level implementation parameters. The impasse formed by signaling channels appears to be fundamental to object-based computing, and is indeed one of the main issues that we address in this dissertation.

1.2.3 Security Kernels, Architectures, and Trusted Subjects

Are there approaches to building secure systems that overcome the security problems inherent in conventional design methodologies? One answer to this lies in the idea of building secure operating systems with a *security kernel* approach. The lower-level functions in an operating system are generally performed by what is considered to be the kernel (also called the nucleus or core). Similarly, in a secure system the security kernel implements the security mechanisms of the operating system. The security kernel approach is based on (and implements) the concept of a *reference monitor*. The reference monitor, which is the combination of hardware and software, enforces the security policy by providing access control to resources. It utilizes access control information stores in an access control database for this purpose.

The successful application of the security kernel approach is based on the theory that only a small fraction of the total functions in an operating system are needed to enforce security. The motivations to isolate the security functions into a security kernel are many. Isolation makes it easier to protect, modify, as well as verify

these security mechanisms. It also makes it easier to ensure completeness or coverage; i.e., every access to a protected object must pass through the security kernel.

Several systems that employ the security kernel approach also support the notion of *trusted subjects*.¹ The most distinguishing feature of trusted subjects is that they are endowed with certain privileges. Most notably, a trusted process may be allowed to bypass mandatory security controls. Thus a trusted subject could access information at various security levels and be allowed to write-down such information (as it is exempt from the \star -property restrictions of mandatory security).

There has been a certain degree of controversy, uneasiness, and suspicion with the notion of trusted subjects. One of the major drawbacks of utilizing trusted subjects has to do with the greater difficulty in verifying trusted software. The properties and associated proofs of trusted subjects are not as obvious or straightforward as that for the simple security and \star properties. In the remainder of this dissertation, we refer to architectures that use trusted subjects as *trusted-subject architectures*. Architectures that use only single-level (untrusted) subjects are referred to as *kernelized architectures*.

1.2.4 Multilevel System Architectures

In this section we briefly review three multilevel database management system (DBMS) architectures, namely, the kernelized, replicated, and trusted subject architectures. Of these, the first two comprise two of the three architectures identified by the Woods Hole study organized by the U.S. Air Force [Cou83]. These architec-

¹The term "trusted" is used often in the literature to convey one of two different notions of trust. In the first case, it conveys the fact that something is trusted to be correct. In the second case, we mean that some subject is exempted from mandatory confidentiality controls; in particular the simple-security and \star -properties in the Bell-Lapadula framework. It is the latter sense of trust that we refer to here.

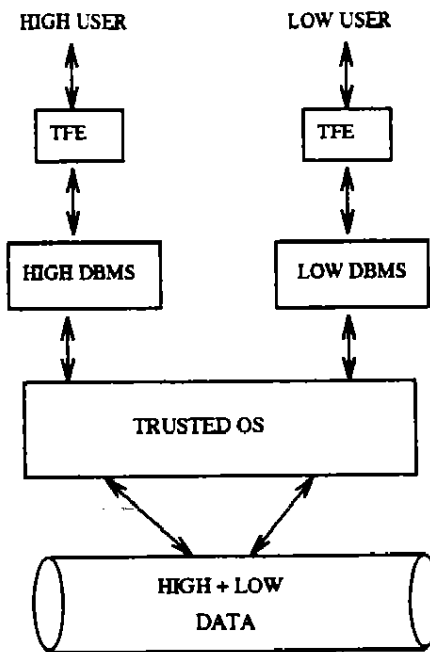


Figure 1.2: A kernelized multilevel system architecture

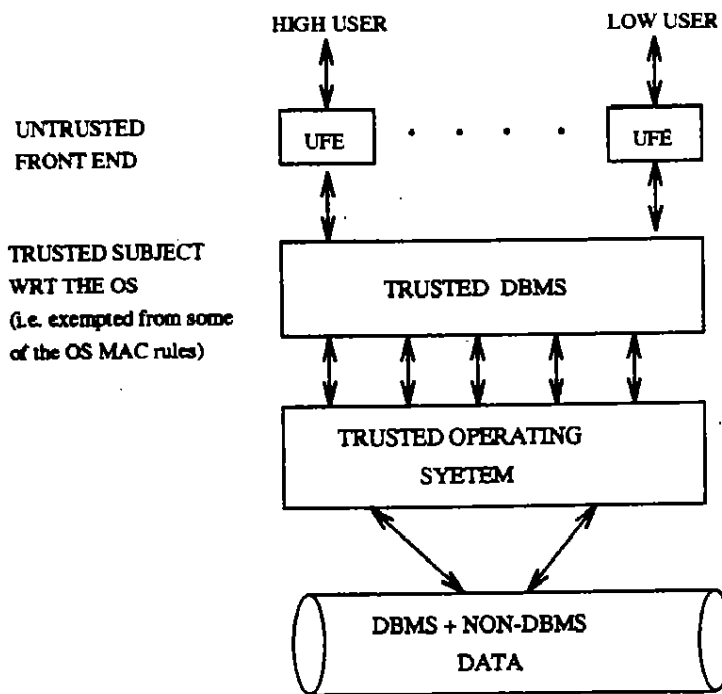


Figure 1.3: A multilevel system architecture with trusted subjects

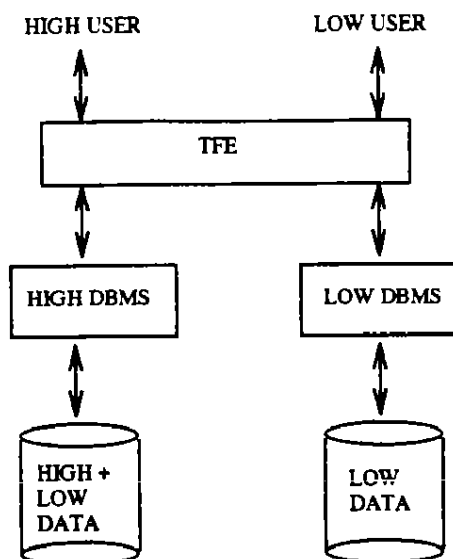


Figure 1.4: The replicated multilevel system architecture

tures were motivated by the need to build multilevel secure DBMS's from existing untrusted DBMS's. The trusted subject architecture on the other hand, requires one to build a multilevel DBMS from scratch.

The basic features of the kernelized architecture are shown in figure 1.2. Most noticeable is the fact that there exists an individual DBMS for every security level. A user cleared to level high, interacts with the high DBMS through a trusted front end. Existing DBMS's can be directly incorporated with minimal modification since they have to manage only single-level data at their levels. These single-level DBMS's communicate with a trusted operating system that manages both high and low data storage.

The trusted subject architecture differs from the kernelized one in that we no longer have single-level DBMS's for every level, as shown in figure 1.3. Instead, there exists a single (trusted) DBMS that incorporates multilevel trusted subjects. Recall that such a subject is exempt from the mandatory access control rules enforced by

the operating system.

In contrast to the kernelized and trusted subject architectures, the distinguishing feature of the replicated architecture shown in figure 1.4 is the existence of separate DBMS's for every level and the replication of low data at the higher level DBMS's. The high assurance of this architecture stems from the fact that the DBMS's are physically isolated and a subject cleared to a level, say l , is allowed to access only the database for level l , and can obtain all the data needed at the local DBMS location.

1.3 Multilevel Security in Object-based Computing

In this section, we discuss the integration of multilevel security in object-based systems. In attempting such an integration, are there fundamentally different approaches to providing access control and authorization in object-based systems? Keefe [Kee90] has provided a useful framework and categorization of existing object-based security models and access control approaches. Namely, these include *behavior-based*, *structure-based*, and *message-based* approaches.

An object models the behavior of an entity through the methods supported by its interface. Accordingly, a behavior-based approach to multilevel security specifies and enforces access control in terms of method invocations. Thus the access control problem essentially reduces to the question: Is subject S allowed to invoke method M on object O ? Access control and rights is no longer seen in terms of read and write operations; rather at the higher and semantic level of abstract operations. The idea of behavior-based access control meshes well with the notion that objects are instances of abstract data types. Unfortunately, the semantic and abstract nature of this approach gives no clue on how to construct a more concrete and implementable model.

In the structure-based approach, access control is no longer seen in terms of

the semantics of methods or abstract operations. Rather, it is based on mediating primitive read and write operations issued by methods. The mediation of reads and writes essentially reduces this approach to an interpretation of the Bell-LaPadula (BLP) model for objects. Access to portions of object states by subjects, is governed by the simple-security and \star -properties of BLP.

The behavior-based and structure-based approaches reflect the bias which systems have in supporting behavioral and structural features. Dittrich has provided a useful taxonomy for object-oriented databases [DHP89], and observes that behaviorally object-oriented systems tailor all mechanisms to emphasize the support of abstract data types. Structurally-oriented systems emphasize the access and manipulation of complex and nested object structures. Fully object-oriented systems provide both behavioral and structural features.

A third approach, referred to hereafter as the message or flow-based approach, is based on the central notion that in the object-based model, objects communicate with each other solely through messages. Since objects are encapsulated units, it follows that security can be enforced by controlling the exchange of messages. When a message is sent, the classifications of the sender and receiver objects are used to determine if an illegal information flow will take place. The message is delivered to the receiver only if the resulting information flow does not violate the security policy. It is important to note that there is no attempt to analyze the semantics (i.e., message type) or contents of the message itself.

How does the message-based approach compare with the behavior-based and structure-based approaches? In some sense, by ignoring the semantics of messages the message-based approach appears to be less rich than the behavior-based one. Nevertheless, it offers the advantage of meshing well with the object-based model of computing. Thus it has wide applicability in providing security for systems ranging

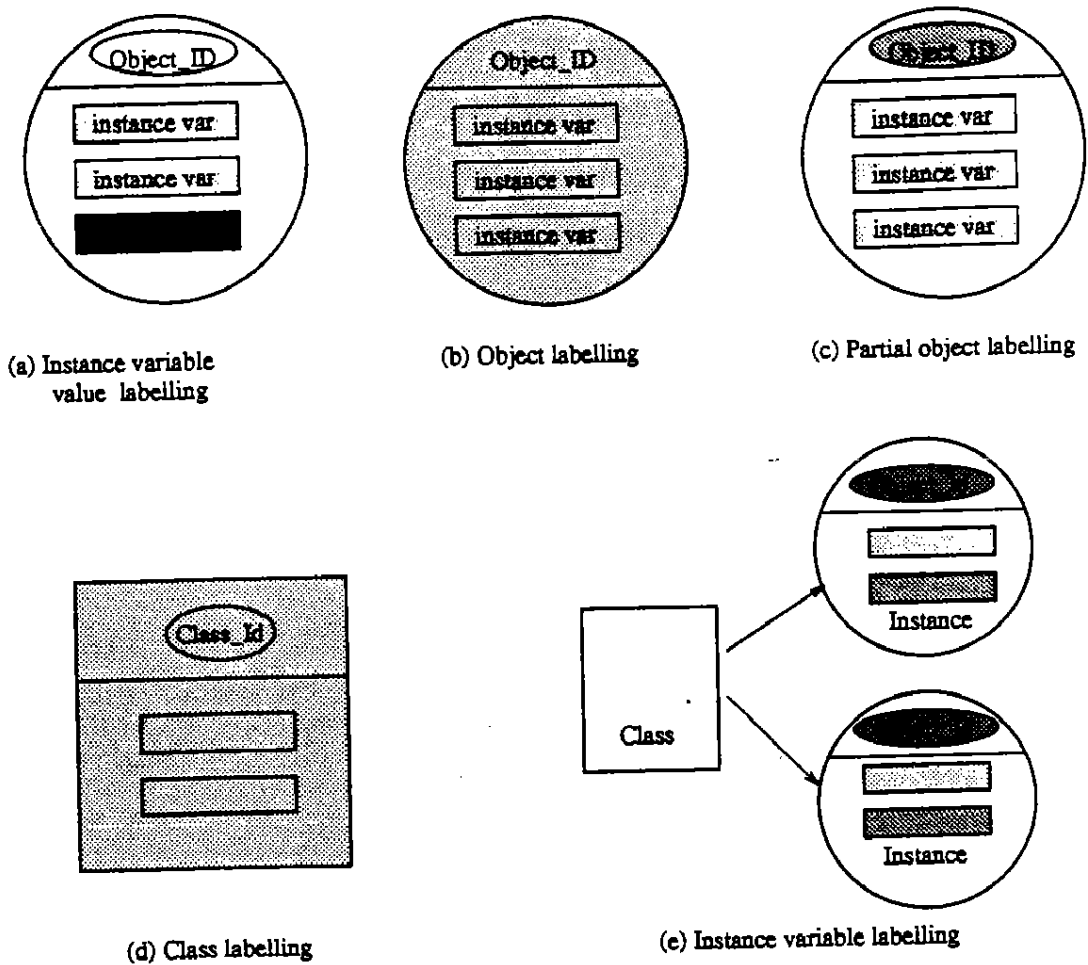


Figure 1.5: Object classification granularities

from message and object-based operating systems to object-oriented databases. A further advantage of the message-based approach is that it lends itself easily to an implementation. It is conceptually simple and elegant to enforce security by mediating messages at a central point such as the trusted computing base. When compared to structural approaches, the message-based approach incurs less overhead since access control need not be enforced on the many primitive read and write operations that can be issued by methods.

Another issue that arises with the integration of multilevel security and object

notions, is that of object classification granularity. What is the basic size of the unit that should be classified independently from other units? If the granularity is larger than an object, the system may offer good performance. However, from a modeling standpoint this may not be flexible enough to classify the objects accurately and naturally. If the granularity is very fine and smaller than an object, the system may offer great modeling flexibility, although at the price of performance.

Figure 1.5 illustrates five different approaches and associated granularities for object classification discussed in [Kee90]². In the first approach, referred to as instance variable value labeling, every instance variable or element of an object is assigned an independent security level. In the second approach, called object labeling, the entire object is uniformly classified. A variation of this second approach yields a third one called partial object labeling, in which all the instance variables share a common uniform classification which may be different from the rest of the object. The fourth approach called class labeling, results in every object instantiated from a class, to be object labeled with a single common classification. The last approach referred to as instance variable labeling, requires the classification of the object-identifiers for all the instance objects instantiated from a class, to be the same. Further, all the instance variables are required to have the same label, or labels that fall within some prespecified range.

In summarizing this section, we note that the approaches to authorization, and classification granularity, are related as they influence each other. The primitive read and write operations always apply to the values of individual instance variables of an object. Thus instance variable value labeling lends itself to a structure-based access control. On the other hand, if message or flow-based access control is used, it is more natural to classify objects uniformly. On receipt of a message, the corresponding

²It is of course possible to enumerate other combinations and possibilities.

method invoked is generally given access to the entire object state within the object boundary. Thus object labeling appears to be more suitable.

1.4 Summary of Previous Work

In this section we review some of the existing proposals to integrate multilevel security in object-based and object-oriented systems. The motivation for most of these efforts have come primarily from object-oriented databases.

A behavioral approach to access control is pursued in [MH89]. A subject S is allowed to invoke a method M on an object O , if a relationship exists between the subject, object, and the method. The security policy is expressed as a set of such relationships.

Structure-based approaches are mentioned in [KTT88, ML92, Lun90, Thu89a, Thu89b]. All these approaches accommodate fine-grained classification granularity by supporting multilevel objects (objects where each component is classified independently). Thus access to components of objects is governed by mandatory access control rules. These models also consider illegal information flows that could occur from classes to instances, as well as through inheritance along the class hierarchy. In order to prevent such flows, these models require a number of constraints to be maintained. For example in [Thu89b] it is required that the classification of an instance of a type dominate the classification of the type. In [Lun90], the hierarchy property requires that the classification of a subclass dominate the classification of the parent superclass. This ensures that information flow along the class hierarchy is always upwards in the security lattice.

Flow or message-based approaches to multilevel security were initially proposed in [BTMD89, CVW⁺88] for message-based secure operating systems. Similar ideas for databases and information systems are mentioned in [MO87, TC89, JK90].

The underlying theme in all these proposals is to enforce security by mediating message flow between objects. The model in [JK90], referred to hereafter as the message filter model, calls for a message filter component to filter messages. The message filter here is the analog of the reference monitor.

Our discussion above has been intentionally brief as the focus in this dissertation is on a very specific issue: the support for write-up actions by sending messages upwards in the security lattice. While the modeling flexibility and ease of implementation of the above proposals and models vary, the support for write-up actions is generally absent, or if present, not worked out in sufficient detail.

The models in [ML92, Thu89b] explicitly prohibit write-up actions. The original message filter proposal in [JK90] allows messages to be sent upwards in the security lattice and places no restrictions on write-up actions. However, it does not address the details and complications involved in doing this. The model in [ML92] also allows messages to be sent upwards in security levels. As in the message-filter model, the authors note that the actual replies to such messages cannot be returned to the lower level senders, and hence have to be substituted with innocuous NULL or NIL messages. However, the following observation by the authors is significant :

“Note that the system, rather than the higher-level subject, should determine the time it takes to deliver the null value, otherwise a timing channel will exist. The underlying TCB is responsible for this protection.”

But how will the TCB provide such protection? What are the architectural requirements? In providing such protection, will integrity be compromised? It is precisely answers to these specific questions that are pursued in this dissertation.

1.5 Organization of Thesis

	Trusted Subject	Kernelized	Replicated
Integrity	Serial correctness	Serial correctness	Serial correctness Final-state-equivalence
Confidentiality	Session manager is noninterfering		

Table 1.1: Summary of main results

The major results of this dissertation cast in terms of the various theorems are summarized in figure 1.1. In the kernelized and replicated architectures we need to show that our solutions preserve integrity. In the trusted subject architecture, we need to show that in addition to integrity, our solutions do not introduce any confidentiality leaks.

The rest of this dissertation is organized as follows. Chapter 2 motivates the main problem addressed in the dissertation. Chapter 3 is a quick overview of the message filter security model, while chapter 4 presents our asynchronous computation model. Chapters 5, 6, and 7 discuss the implementation of our ideas in trusted subject, kernelized, and replicated architectures, respectively. Chapter 8 discusses inter-session synchronization schemes, and chapter 9 summarizes the dissertation and also highlights some future directions for research.

Chapter 2

Motivation and Problem Statement

In this chapter we discuss the main problem addressed in this dissertation; the support of secure and efficient write-up actions in multilevel object-based computing. We begin with two motivating examples to illustrate the usefulness of write-up operations. We then discuss the various confidentiality, integrity, and performance tradeoffs involved in supporting write-up operations.

Recall that multilevel security mandates information flow to be always upwards in the security lattice. However, there is no reason to disallow a low-level subject from writing-up to higher levels, as the information flow is from low to high (upwards in the lattice). Such write-up actions are natural and very useful in modeling many applications (as we will illustrate shortly). Unfortunately, an analysis of multilevel systems, particularly databases, would reveal that support for write-up actions is generally absent, or at best weak and ad-hoc. These systems typically implement a restricted version of the BLP \star -property that allows writes only at the level of the subject requesting the write operation.

We may partly attribute the above reluctance in supporting write-up actions to a fundamental conflict between confidentiality and integrity [MA91]. This is because the requirements to enforce integrity constraints often result in confidentiality being compromised. Conversely, guaranteeing confidentiality may require tolerating lower degrees of integrity. In conventional databases such as relational systems, the effect of

arbitrary blind write-up operations on integrity is unpredictable and uncontrollable. Thus, there always exists the potential for a low-level subject to obliterate higher-level data. Now in the object-based framework, we cannot foresee a similar threat to integrity. Why is this? Because if objects can communicate solely through messages, the properties of encapsulation and information hiding will ensure that an object state is updated only in controllable ways. On receiving a message from a lower level, a high-level object can exercise complete control over how, and if, its state should be updated. It may choose to reject the message request by not invoking the corresponding method. If the message is accepted, the method invoked has precise semantics known a priori.

Thus the objective of supporting integrity preserving write-up actions seems tenable within the object-based framework. Unfortunately, this convenience does come at a price. Ironically, the very feature of objects (the ability to incorporate well-defined semantics with operations) poses confidentiality leaks. In the object framework, operations are no longer primitive read's and write's, but complex and abstract, and taking varying amounts of processing time. As we will elaborate shortly, this can cause signaling channels when write-up actions are issued [STJ92].

In retrospect, traditional models such as Bell-LaPadula (BLP) address security issues primarily within the realm of multilevel operating systems and not databases. Thus BLP does not associate much semantics with operations; rather, it views operations as basically primitive read's and write's on memory segments. Also, the BLP view of write-up actions boils down to blind writes (i.e., modification of existing higher-level data is not permitted while its overwriting is permitted). Thus a request from a low-level subject to debit or credit a high-level bank account, cannot be handled by the BLP framework. These restrictions, or should we say limitations, have enabled BLP and its derivatives to abstract away the problem of signaling channels

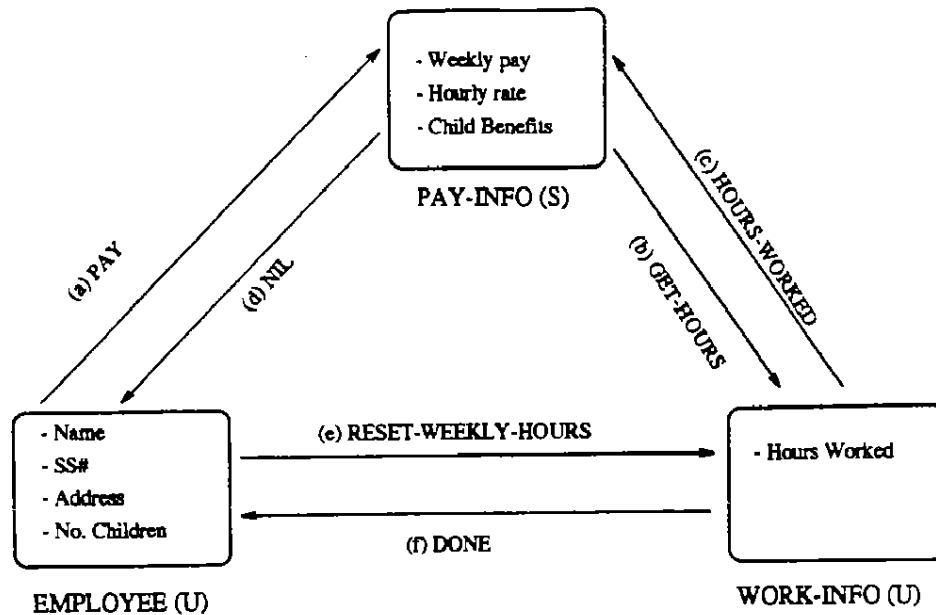


Figure 2.1: Objects in a payroll database

during write-up. If write-up actions are primitive, and thus implemented by machine language instructions such as STORE, it may be reasonable to assume that STORE operations will take a fixed amount of time independent of the data and address. In reality this assumption is only an approximation to what happens in modern computers. Paging, caching, bus contention, and CPU load, to name a few, modulate the time taken to complete operations such as STORE. Depending on the implementation, such variations can be exploited for timing channels and this has been recognized by the security community [Cip90].

2.1 Motivating Applications and Examples

We now motivate the usefulness of write-up operations accomplished by sending messages up in the security lattice, with two examples.

2.1.1 Payroll Processing Example

Consider a database for payroll applications, that has three objects: EMPLOYEE (Unclassified), WORK-INFO (Unclassified), and PAY-INFO (Secret), with the attributes shown in figure 2.1. Every object is assigned a single level. Weekly payroll processing is initiated by the lower level EMPLOYEE object with the sending of the (a) PAY message to the higher level PAY-INFO object. As the receiver is at a higher level than the sender, an innocuous NIL reply is returned by the message filter (as mandated by the message filtering algorithm, which will be discussed in the next chapter). On receiving the PAY message, the method in PAY-INFO sends a read-down message (b) GET-HOURS, to the lower level WORK-INFO object in order to retrieve the hours worked. This information is retrieved and returned in the reply message (c) HOURS-WORKED. Finally, the accumulated hours for the week is reset to zero by the message (e) RESET-WEEKLY-HOURS.

Another scenario for write-up arises when the child-benefits an employee is eligible for needs to be updated due to an increase in the number of children. Such an update is most efficiently accomplished by a trigger fired in the lower level EMPLOYEE object when the NO-CHILDREN attribute changes. The trigger would result in the sending of a message with the value of number of children, NO-CHILDREN, as a parameter to the higher level object PAY-INFO. The alternative to such a write-up would be that the PAY-INFO object scan the corresponding EMPLOYEE object for such changes, whenever the payroll is computed. However, this alternative imposes a significant performance cost for slow-changing information such as NO-CHILDREN. Further, incorporating such monitoring capabilities into methods lowers the reuse potential of the corresponding objects and classes. We elaborate on these issues in the next example.

2.1.2 Situation Assessment Example

Our second example is in the domain of situation assessment. Figure 2.2 illustrates a tactical situation with four objects POSITION-UPDATE (Confidential), TARGET-LOCATOR (Secret), TARGET-TO-SHIP-DISTANCE (Secret), and ACTION-UPDATE (Top-secret). The object POSITION-UPDATE receives and records periodic updates of the position of an AWACS aircraft. After recording the position, it is reported through the message REPORT-POSITION to the object TARGET-LOCATOR. The object TARGET-LOCATOR locates any targets near the reported position and further sends two messages CALC-DISTANCE and DETERMINE-ACTION. The first message is received by the object TARGET-TO-SHIP-DISTANCES which calculates the distances between ships in the fleet and the identified targets, and in turn reports these distances with the message REPORT-DISTANCE to the object ACTION-UPDATE. The second message DETERMINE-ACTION sent by TARGET-LOCATOR is also received by the object ACTION-UPDATE. Finally, on receiving the DETERMINE-ACTION and REPORT-DISTANCE messages, the object ACTION-UPDATE selects one or more ships or other attack vehicles that are within striking range of the target, and initiates some action.

In this example, the messages REPORT-POSITION, REPORT-DISTANCE, and DETERMINE-ACTION, lead to write-up actions. One could always argue whether the above application could be implemented with read-down operations. But we observe that application areas such as situation assessment, battle management, network monitoring, and process control, have sparked a great interest in active databases. Why? Because in these applications the processing steps involve the monitoring of conditions, and the invocation of time-constrained actions when certain conditions come true. In our example, when the AWACS aircraft crosses over a new coordinate, an update of its position is triggered. This update in turn triggers other

processing steps. Implementing these steps with read-down operations would require extensive polling of low level object states by higher level objects.

The difficulty with polling is that it is not very feasible to determine an appropriate polling window (interval) especially when the interval between triggered events is unpredictable and not constant. An inaccurate polling window resulting from guesswork, can have drastic consequences. For example, if the higher level object TARGET-LOCATOR polls the lower object POSITION-UPDATE for updates on the aircraft's position too slowly, it might miss some vital positions that were covered by the aircraft. Clearly, this can result in potential targets escaping detection and identification. On the other hand, if the POSITION-UPDATE object is polled too frequently, it may be flooded with repetitious read-down requests that waste resources and affect performance. In fact, the object may be so overwhelmed with these requests that it may not be able to keep up with the the useful and timely position updates from the aircraft. This again, can result in many targets being missed.

2.2 Write-Up and Confidentiality, Integrity, and Performance Tradeoffs

Having motivated the need for write-up actions, we now discuss the issues, conflicts, and trade-offs involved in supporting such actions in multilevel object-based computing environments. In particular, we highlight the trade-offs between confidentiality, integrity, and performance.

Going back to the basics, let us see what happens when a message is sent to a higher security level for the purpose of initiating some write-up action. Figure 2.3 depicts a message g_1 sent from a sender object O_1 to a receiver object O_2 , with the receiver classified at a higher security level. Now in synchronous communication mode, the sender method t_1 in object O_1 is effectively suspended once the message

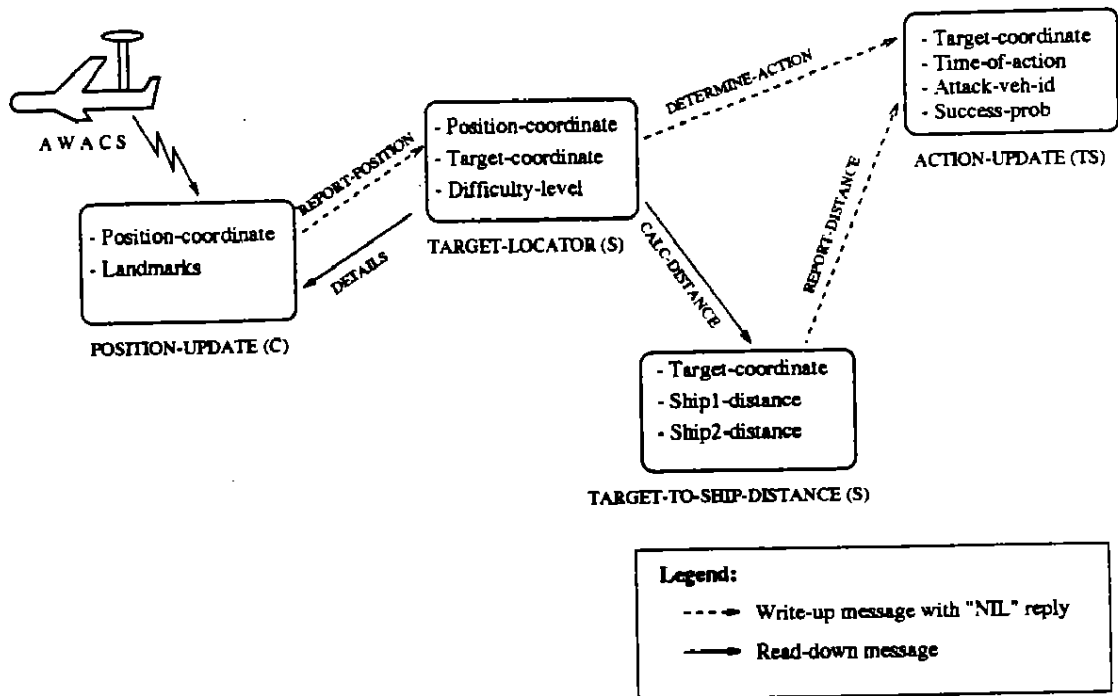


Figure 2.2: Write-up in situation assessment

g_1 has been sent. The receipt of g_1 by O_2 will result in the invocation of a receiver method t_2 .

In the multilevel context, it is clear that the contents of the actual reply from t_2 cannot be returned to the lower level receiver method (or object), for doing so would lead to an illegal information flow (in fact, the security kernel and mandatory security rules would prevent such an attempt). A conceptually simple solution would be to arrange for an innocuous reply such as a NIL to be substituted and returned by the kernel. This does not result in any direct illegal information flow from the higher level object, as no information based on its contents is made known to the lower level sender. However, it turns out that the very timing of such a reply has broad implications on confidentiality, integrity, and performance issues.¹

¹Strictly speaking, we do not require any reply (NIL or other) to be returned to a suspended

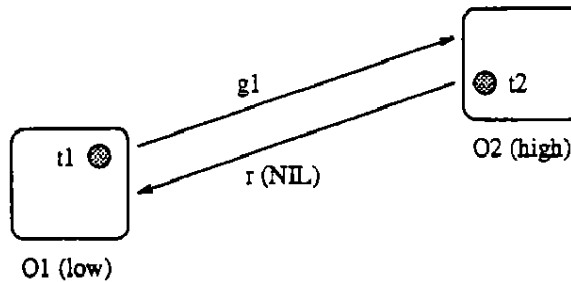


Figure 2.3: A writeup message and its reply

To elaborate on the above, consider the following alternate ways to deal with message replies:

- **Option 1:** Return a NIL reply on completion of the method in the receiver object;
- **Option 2:** Return the reply independent of the termination of the receiver method in one of the following ways:
 - **Option 2a:** Return the NIL reply after some constant time interval that represents an upper bound for completion times;
 - **Option 2b:** Return the reply after some random delay;
 - **Option 2c:** Return the NIL reply instantaneously.

With the first option, we have a sequential synchronous execution of methods governed by remote procedure call semantics. Now the time taken for the receiver method to complete is by no means constant or predictable. This can happen for example if

sender. The real issue is not so much the timing of the reply, but rather when a suspended method should be resumed. This can be done irrespective of whether a reply is received. Thus it is only for uniformity and ease of exposition, that we assume the receipt of the NIL reply as a logical point to resume a suspended sender method.

the receiver method sends more messages and completes other subtasks before terminating. Further, the receiver method in the higher level object has complete control over its termination. Thus by varying the completion times, the receiver method can modulate the timing of the reply, and this opens up the potential for a signaling channel.

The second set of options attempts to eliminate the above signaling channel by making it impossible for the delivery of the NIL reply to be modulated by a higher level method. Option 2a imposes a heavy performance penalty whenever the receiver method has terminated and the sender remains unnecessarily suspended, waiting for the constant time interval to elapse. If we adopt option 2b, by randomizing the delay before returning the reply, we are faced with a tradeoff between performance and integrity. This is because if the reply is returned well after the termination of the receiver method, we are again unnecessarily holding up the sender method. On the other hand, if we return the reply too early, that is, before the receiver method has terminated, we have to deal with the concurrent execution of methods.

Concurrent executions introduce synchronization problems that can affect the integrity of the database. In particular, it is essential that the concurrent executions guarantee equivalence to a sequential execution, as in the first option. In other words the updates and reads issued by concurrent methods should have the same effect as when the methods are executed synchronously. When such equivalence can be guaranteed, we say that the concurrent execution of the methods (computations) preserve *serial correctness*. In the next section, we give some concrete examples that illustrate how concurrency can affect serial correctness. Note that this requirement of preserving serial correctness is entirely dictated by integrity considerations. From a confidentiality viewpoint, there is no need to synchronize these concurrent executions.

We now illustrate a scenario on how the integrity of the payroll database

(see figure 2.1) can be compromised. In this scenario, the application semantics and requirements called for a synchronous execution of methods but the resulting execution was concurrent (asynchronous). Now a sequential synchronous execution will lead to the message sequence *a, b, c, d, e, f*; while a concurrent execution may produce the sequence *a, d, e, f, b, c*. When weekly payroll processing is initiated by the sending of the PAY message from the lower level EMPLOYEE (U) object to the higher level PAY-INFO (S) object, a NIL reply is returned to object EMPLOYEE and the suspended method in EMPLOYEE resumes execution. Now it is possible for the RESET-WEEKLY-HOURS message which resets the hours worked to zero, to be received and processed by object WORK-INFO before the message GET-HOURS. Thus the message GET-HOURS will retrieve the reset hours as opposed to the actual accumulated hours, resulting in an erroneous calculation of the weekly pay.

We will demonstrate later in chapter 4, how the required integrity can be achieved by the use of a multiversioning scheme that synchronizes concurrent actions on objects so as to guarantee serial correctness. To see how the multiversioning scheme applies to above the payroll example, the (e) RESET-WEEKLY message would result in the creation of a new version of object WORK-INFO with the reset hours. However, an earlier version of object WORK-INFO that existed before the method in PAY-INFO was invoked, is used to process the (b) GET-HOURS message. Serial correctness is now ensured as the GET-HOURS message now retrieves the intended weekly accumulated hours as in the sequential synchronous execution.

Finally, option 2c above calls for replies to be returned instantaneously. We thus no longer incur the performance penalty that is possible with options 2a and 2b. However, we still have to address the integrity issue, as concurrent computations are now inevitable.

How do the various architectures impact our choice of one of the above options?

Option 1 is inherently insecure in trusted subject architectures. Option 1 is further not implementable in a kernelized architecture as the \star -property prevents information flow from a higher level to a lower one, by disallowing write-downs (recall that only a trusted subject is allowed to indulge in such write-downs). Such write-down operations are required to inform lower sender methods of the termination of higher level receivers. Option 2a and 2b are implementable in a kernelized architecture but at the cost of performance and integrity. Option 2c needs to address the integrity issue just as option 2b, but offers better performance than the latter, although as with option 2b, this comes at the cost of managing concurrency.

In summary, synchronous RPC-based write-up actions are not secure in trusted subject architectures, and not implementable in kernelized architectures. Thus our only viable choice is to implement message passing and write-up actions in multilevel environments in an asynchronous fashion. The challenge, and our focus, then is to provide the desired RPC-based semantics for asynchronous abstract write-up actions. This requires appropriate synchronization mechanisms, which themselves have to be secure and implementable.

Chapter 3

Message Filtering and Mandatory Security Enforcement

In this chapter we introduce the message filter object-oriented security model. This model is used to enforce basic mandatory access control among objects. We begin with a discussion of the message filtering algorithm and message filtering functions. This is followed by a discussion of architectural and other implementation issues. In particular, we illustrate what it takes to map an abstract specification of the filtering functions to an executable one.

3.1 The Message Filter Model

Objects and messages constitute the main entities in the message filter model. As far as the security model is concerned, an entire object is classified at a single level. Modeling flexibility is not lost due to this as a user may model multilevel entities. The multilevel entities form a conceptual schema that is broken down into an implementation schema of single-level objects [JK90]. Messages are assumed, and required to be, the only means by which objects communicate and exchange information. Thus the core idea is that information flow be controlled by mediating the flow of messages. Consequently, even basic object activity such as access to internal attributes and object creation, are to be implemented by having an object send messages to itself (we consider such messages to be primitive messages).

The message filter is the analog of the reference monitor in traditional access-mediation models. The message filter takes appropriate action upon intercepting a message and examining the classifications of the sender and receiver of the message. It may let the message pass unaltered or interpose a NIL reply in place of the actual reply; or set the status of method invocations as restricted or unrestricted (explained later).

3.1.1 The Message Filtering Algorithm

The message filter algorithm is given in figure 3.1. (In this and other algorithms, the % symbol is used to delimit comments.) Cases (1) through (4) deal with abstract messages, which are processed by methods. Cases (5) through (7) deal with primitive messages, which are directly processed by the security kernel. In case (1), the sender and receiver are at the same security level, and the message g_1 and its reply are allowed to pass. In case (2) the levels are incomparable and thus the filter blocks the message from getting to the receiver object, and further injects a NIL reply. Case (3) involves a receiver at a higher level than the sender. The message is allowed to pass but the filter discards the actual reply, and substitutes a NIL instead. (As we have argued, the timing of this NIL reply is a critical consideration.) In case (4), the receiver object is at a lower level than the sender and the filter allows both the message and the reply to pass unaltered.

The cases (1) through (4) that we have seen so far deal with abstract messages. However abstract messages will eventually lead to the invocation of primitive messages. These include **read**, **write** and **create** (cases (5) through (7)).¹ Now **read** operations always succeed, while **writes** succeed only if the status of the method in-

¹The **delete** operation has not been directly incorporated into the model. It can be viewed as a particularly drastic form of **write** and is subject to the same restrictions.

```

% let  $g_1 = (h_1, (p_1, \dots, p_k), \tau)$  be the message sent from  $o_1$  to  $o_2$  where
%  $h_1$  is the message name,  $p_1, \dots, p_k$  are message parameters,  $\tau$  is the return value
if  $o_1 \neq o_2 \vee h_1 \notin \{\text{read, write, create}\}$  then case
% i.e.,  $g_1$  is a non-primitive message
(1)  $L(o_1) = L(o_2)$  : % let  $g_1$  pass, let reply pass
      invoke  $t_2$  with  $rlevel(t_2) \leftarrow rlevel(t_1)$ ;
       $\tau \leftarrow$  reply from  $t_2$ ; return  $\tau$  to  $t_1$ ;
(2)  $L(o_1) <> L(o_2)$  : % block  $g_1$ , inject NIL reply
       $\tau \leftarrow$  NIL; return  $\tau$  to  $t_1$ ;
(3)  $L(o_1) < L(o_2)$  : % let  $g_1$  pass, inject NIL reply, ignore actual reply
       $\tau \leftarrow$  NIL; return  $\tau$  to  $t_1$ ;
      invoke  $t_2$  with  $rlevel(t_2) \leftarrow \text{lub}[L(o_2), rlevel(t_1)]$ ;
      % where lub denotes least upper bound
      discard reply from  $t_2$ ;
(4)  $L(o_1) > L(o_2)$  : % let  $g_1$  pass, let reply pass
      invoke  $t_2$  with  $rlevel(t_2) \leftarrow rlevel(t_1)$ ;
       $\tau \leftarrow$  reply from  $t_2$ ; return  $\tau$  to  $t_1$ ;
end case;

if  $o_1 = o_2 \wedge h_1 \in \{\text{read, write, create}\}$  then case
% i.e.,  $g_1$  is a primitive message
% let  $v_i$  be the value that is to be bound to attribute  $a_i$ 
(5)  $g_1 = (\text{read}, (a_j), \tau)$  : % allow unconditionally
       $\tau \leftarrow$  value of  $a_j$ ; return  $\tau$  to  $t_1$ ;
(6)  $g_1 = (\text{write}, (a_j, v_j), \tau)$  : % allow if status of  $t_1$  is unrestricted
      if  $rlevel(t_1) = L(o_1)$ 
        then [ $a_j \leftarrow v_j$ ;  $\tau \leftarrow$  SUCCESS]
        else  $\tau \leftarrow$  FAILURE;
      return  $\tau$  to  $t_1$ ;
(7)  $g_1 = (\text{create}, (v_1, \dots, v_k, S_j), \tau)$  : % allow if  $t_1$  is unrestricted relative to  $S_j$ 
      if  $rlevel(t_1) \leq S_j$ 
        then [CREATE  $i$  with values  $v_1, \dots, v_k$  and  $L(i) \leftarrow S_j$ ;
               $\tau \leftarrow i$ ]
        else  $\tau \leftarrow$  FAILURE;
      return  $\tau$  to  $t_1$ ;

end case;

```

Figure 3.1: Message filtering algorithm

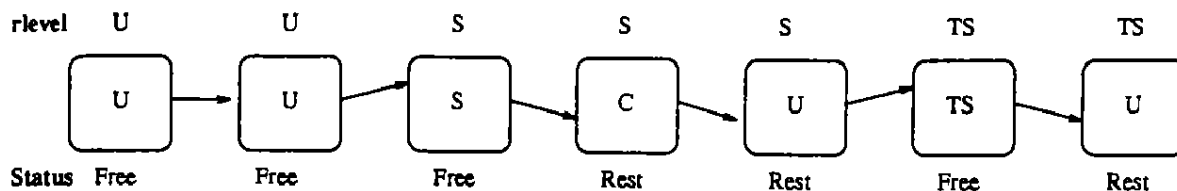


Figure 3.2: Restricted methods in a chain

voking the operation is unrestricted. Thus if a message is sent to a receiver object at a lower level (as in case (4)), the resulting method invocation will always be restricted and the corresponding primitive `write` operation will not succeed. This will ensure that a write-down violation will not occur. Finally, the `create` operation allows the creation of a new object at or above the level of the method invoking the `create`.

3.1.2 Restricted Methods and Invocation Trees

We now revisit the notion of restricted method invocations alluded to earlier. To start with, observe that in cases (1), (3), and (4) of the filtering algorithm, the method in the receiver object is invoked at a security level given by the variable *rlevel*. In other words, the method body is executed by a subject (process) running at level *rlevel*. The intuitive significance of *rlevel* is that it keeps track of the least upper bound (lub) of all objects encountered in a chain of method invocations, going back to the root (first object and method) of the chain. The value of *rlevel* needs to be computed for each receiver method invocation. In cases (1) and (4) the *rlevel* of the receiver method is the same as the *rlevel* of the sender method. In case (3), *rlevel* is the least upper bound of the *rlevel* of the sender method, and the classification of the receiver object. These ideas are illustrated in figure 3.2 for a method chain starting at an unclassified object.

Let us see how *rlevel* implements the notion of restricted method invocations so as to prevent write-down violations. Observe that if t_i is a method invocation in

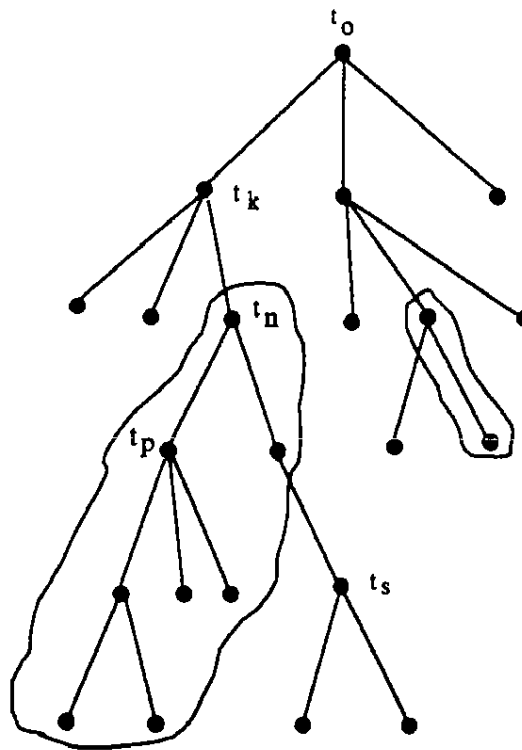


Figure 3.3: A tree with restricted subtrees

object o_i then $rlevel(t_i) \geq L(o_i)$. We say that a method invocation t_i has a *restricted status* if $rlevel(t_i) > L(o_i)$. When t_i is restricted, it can no longer update the state of the object o_i , it belongs to (i.e., its home object). Why? This is because if a method invocation is assigned a level given by $rlevel$, then information classified at $rlevel$ is now available to (flowing into) the method from one or more objects classified at $rlevel$ and encountered earlier in the chain. For example, such information flow could occur through message parameters. To illustrate, consider the secret (S) and confidential (C) objects in the chain in 3.2. The secret object sends a message to the confidential one, resulting in a restricted method invocation in the latter. Secret information could be passed in message parameters to the lower level (confidential) receiver. If information derived from such parameters is used by the receiver method

to update its home object, a write-down violation would occur. Hence the method invocation in the lower level (home) object is restricted.

We can visualize chains of method invocations as belonging to a tree such as in figure 3.3. Restricted method invocations in these chains now show up as restricted paths and subtrees. In figure 3.3, t_k represents a method in object o_k that sent a message, and t_n represents the method invoked in the receiver object o_n . The method t_n is given a restricted status as $L(o_n) < L(o_k)$. The children and descendants of t_n will continue to have a restricted status till such point as t_s . At t_s , the restricted status is removed since $L(o_s) \geq L(o_k)$ and a write by t_s to the state of o_s no longer constitutes a write-down violation.

3.2 Implementing Message Filtering

Having given an introduction to the message filter model, we now turn our attention to implementation. In particular, we discuss what it takes to map an abstract specification of the filtering functions (as given in the filtering algorithm) to an executable one [STJ91, TS94a]. We begin by elaborating some architectural considerations.

3.2.1 System Layering and the Security Perimeter

In architectural terms, how should systems be structured and layered to incorporate and enforce mandatory security and message filtering? A good and logical starting point for our investigations is the architecture of existing object-based systems. This is because we want our solutions to be cost-effective and thus fit within existing implementation frameworks.

An analysis of many prototypes and proposals for object-oriented database systems such as GEMSTONE, IRIS, ORION (to name a few) would reveal a common

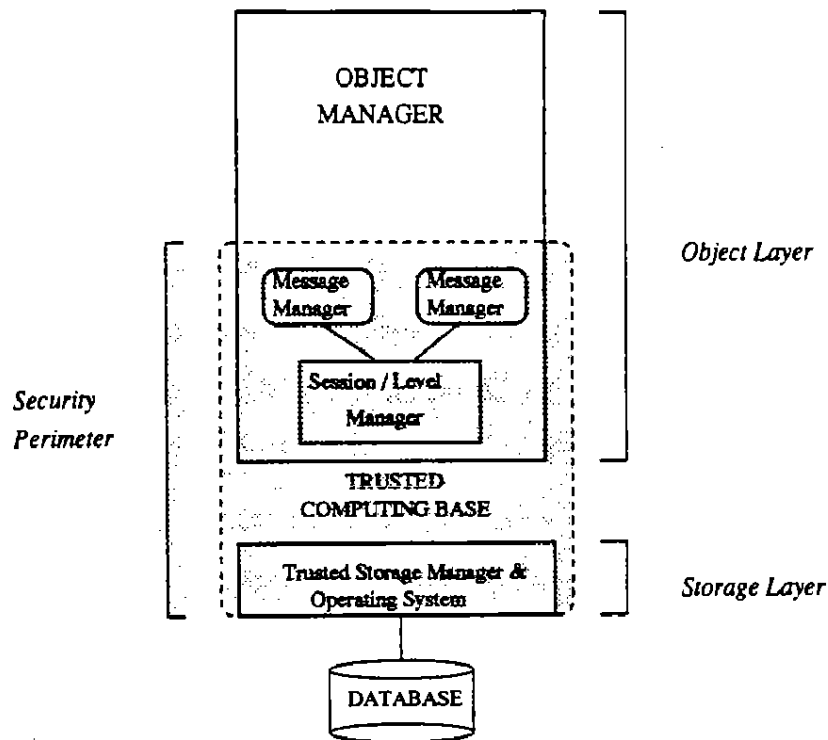


Figure 3.4: A layered architecture with TCB

architectural structure; a demarcation into a lower *storage layer* and an *object layer* on top of the storage layer. For example, in GEMSTONE the lower layer is referred to as STONE, while the object layer consists of GEM processes. The lower storage layer essentially interfaces to the operating system and file system primitives, and is responsible for the management (i.e., the read, write, and creation) of typeless chunks of bytes representing objects. Every object (chunk of bytes) is associated and represented by a unique object-identifier. This layer typically does not understand the abstraction of objects or the object-oriented data model, rather sees itself as one that provides basic services to the higher layers such as the object layer.

In contrast to the storage layer, the object layer is not typeless, but rather supports the abstraction of objects as encapsulated and typed units of information. This

layer is thus responsible for implementing the object-oriented data model. Object-oriented concepts such as classes, class-hierarchies, inheritance, as well as message passing lie within the purview of this layer.

Given the above layering structure, how and where will the TCB fit in? Also, what subsets or functions of these layers should lie within the security perimeter? It is clear that some subset of the operating system and the storage layer need to be within the TCB. But what about the object layer? Could we not realize a secure system by having just part of the storage layer in the TCB? If confidentiality were our only objective, the answer to the latter question would be "yes". Mandatory confidentiality can be enforced by the subset of the operating system and storage layer within the TCB. However, integrity is a vital requirement that has to be maintained alongside confidentiality. The maintenance of integrity for objects cannot be done at the lower layer since it does not recognize the abstractions of the object model. Hence, a subset of the object layer needs to be within the TCB for integrity purposes.

Recall that a good design principle for security kernels is to keep its size to a minimum. We thus require that much of the functionality to implement the object-oriented data model be outside the TCB. Thus even support for the notion of objects as units of encapsulation, is provided by the object layer subset outside the TCB. The subset within the TCB implements high assurance and integrity preserving message passing and filtering. This is accomplished through modules called message managers.² A *message manager* process is created dynamically whenever a message is sent upwards in the security lattice and concurrent execution of the sender and receiver methods is required. Once created, it implements the message filter-

²In our further discussions, we use the terms message managers, computations, and methods, interchangeably. However, it is to be understood that message managers are created only as a result of write-up messages. A message manager is thus a concurrent computation executing a chain of methods.

ing algorithm for the chain of methods emanating from such a concurrent receiver method. A message manager is thus a relatively short-lived process, and one that eventually terminates along with the last method in the associated chain. In a trusted subject architecture the level managers that are forked by a single user session are coordinated by a trusted multilevel process called a *session manager*. A *user session* encompasses all the computations (activities) initiated by a user between consecutive logins. In an architecture without trusted subjects (such as a kernelized one), the message managers are coordinated by untrusted single-level processes called *level managers*.

In summary, the message manager and level (or session) manager modules in the object layer need to be within the security perimeter (TCB) so as to ensure serial correctness (mentioned earlier in chapter 2). If the concurrent message managers cannot guarantee this, the integrity of the data in the database could be severely compromised.

3.2.2 An Executable Specification

The message filtering algorithm presented earlier can be thought of as an abstract non-executable specification of the filtering functions. An executable specification, as implemented by a message manager, is given in figure 3.5. The security perimeter of the object layer exports the following operations: **send**, **quit**, **read**, **write**, and **create**. The **read**, **write**, and **create** operations handle primitive messages. The system primitives **send** and **quit** are used by methods to send messages and replies. A stack is used to save the contexts associated with nested message sends. The interface between a message manager and a level or session manager consists of two calls: (1) **fork** issued by a message manager to request creation of a new message manager at a higher level and (2) **terminate** issued by a message manager to its

local level or session manager to terminate itself.

Whenever a message is sent by a method t_1 in an object o_1 to a second object o_2 at the same or lower level (cases (1) and (4)), the message manager saves the message parameters on a new stack frame, suspends execution of t_1 , and begins execution of the method t_2 in object o_2 . When t_2 terminates, the stack is popped and the return value from t_2 is recorded on the stack. The suspended sender method t_1 is then resumed, and it retrieves the return value from t_2 from the top frame of the stack.

When messages are sent to incomparable or higher levels (cases (2) and (3)), a NIL value is recorded on the stack and t_1 is resumed immediately. In case (3) when a message is sent upwards in the security lattice, a message manager issues a **fork** call resulting in concurrent computations (as t_1 is resumed independently of the termination of t_2). The parameters of this call include the level of forking message manager, the level of the forked message manager, a unique fork stamp identifying the start order in the equivalent sequential execution for the forked message manager, and a vector (astamps) of timestamps to process read down requests. Whenever a reply is returned and a message manager finds its stack to be empty, it means that there are no suspended methods waiting to be resumed. The message manager then issues a **terminate** call to its local level manager, to terminate itself. The parameters of the terminate call include the level and fork stamp of the terminated message manager, as well as a timestamp identifying the last written version.

In moving from an abstract to an executable specification, we have so far described how the filter allows and blocks messages, and how return values are set to NIL. Now it remains to show how the notions of *rlevel* and restricted method invocations are implemented. The basic idea is very straightforward. Every message manager (process) is assigned a security level that is equivalent to the *rlevel* assigned in the filtering algorithm, and all methods executed by a message manager run at this

level. The effect of restricted method invocations is now achieved by the enforcement of the standard \star -property in the Bell-LaPadula type security models [BL76]. In other words, whenever a method's status is restricted, its level (and the level of its message manager) will be higher than the object accessed, and the \star -property will prevent any write-down attempts.


```

procedure send( $g_1, o_1, o_2$ )
% let  $g_1 = (h_1, (p_1, \dots, p_k), r)$  be the non-primitive message sent from  $o_1$  to  $o_2$ 
% where  $h_1$  is the message name,  $p_1, \dots, p_k$  are message parameters, and  $r$ 
%  $p$  is the parameter set  $p_1, \dots, p_k$  and  $lmsgmgr$  is the level of the message manager  $t_1$ 
(1)  $L(o_1) = L(o_2)$  : push-stack( $p$ );
                         $t_2 \leftarrow$  select method for  $o_2$  based on  $h_1$ ; execute  $t_2$ ;
(2)  $L(o_1) \sim L(o_2)$  : write-stack(NIL); resume;
% Let  $astamps$  be a vector that is passed to a forked message manager
(3)  $L(o_1) < L(o_2)$  : append-astamps-vector( $astamps, wstamp$ );
                        fork( $lmsgmgr, lub[lmsgmgr, L(o_2)], forkstamp, astamps$ );
                         $wstamp \leftarrow wstamp + 1$ ;
                        write-stack(NIL); resume;
(4)  $L(o_1) > L(o_2)$  : push-stack( $p$ );
                         $t_2 \leftarrow$  select method for  $o_2$  based on  $h_1$ ; execute  $t_2$ ;
end case;
if  $o_1 = o_2 \wedge h_1 \in \{\text{read, write, create}\}$  then case % i.e.,  $g_1$  is a primitive message
(5)  $h_1 = \text{read}$  : if  $L(o_1) = lmsgmgr$  then  $v \leftarrow wstamp$ 
                  else  $v \leftarrow \text{local-stamp}$  ( $L(o_1)$ );
                  read  $o_1$  with version  $\leftarrow \max\{\text{version: version} \leq v\}$ ;
(6)  $h_1 = \text{write}$  : write  $o_1$  with version  $\leftarrow wstamp$ ;
% Let  $o$  be the object-identifier of the new object created at level  $S_j$ 
(7)  $h_1 = \text{create}$  : create  $o$  with  $L(o) \leftarrow S_j$  and version  $\leftarrow wstamp$ ;
                  write-stack( $o$ );
end case;
end procedure send;

procedure quit( $r$ )
  pop-stack;
  if empty-stack then terminate( $lmsgmgr, wstamp, forkstamp$ )
  else [write-stack( $r$ ); resume];
end procedure quit;

```

Figure 3.5: Message manager algorithms for SEND and QUIT

Chapter 4

Asynchronous Computing with RPC Semantics

In this chapter we focus on issues related to concurrency and scheduling within a single user session. We begin by discussing the notion of serial correctness and how this governs the degree of concurrency that can be allowed within a session. Maintenance of serial correctness requires that we capture the serial order of computations. This is done by means of a hierarchical scheme to generate forkstamps. Two extreme scheduling strategies both of which preserve serial correctness, but offer varying degrees of concurrency, are then discussed. Finally we present a framework for the comparative analysis of these and other scheduling schemes.

4.1 Serial Correctness versus Concurrency

In chapter 2 we discussed the synchronization problem caused by concurrent computations and how this can affect serial correctness. To elaborate in more general terms, visualize a set of concurrent computations as a tree such as that shown in figure 4.1. In this figure we see that message manager 1 at the unclassified level has sent messages to one secret object, one top-secret object, and one confidential object in this sequence (we consider message manager 1 to be the ancestor of the three). As these objects are higher in level than unclassified, message filtering has resulted in the creation and concurrent execution of message managers 2, 3, and 4 as children of the root message manager 1.

We can now formally define serial correctness in terms of such a tree.¹

Definition 4.1 *We say a session preserves serial correctness if for any computation c in the session's computation tree, and running at level 1, the following hold:*

1. c does not see any updates (by reading-down) of lower level computations that are to its right², in the tree;
2. For any of c 's ancestor computations a , (i.e., any computation on the path from the root to c) c should see only the latest updates made by a just before a 's child (or c itself) on this path was forked.
3. For any level k that is not the level of an ancestor of c , and $k \leq 1$, c should see the latest updates made by the rightmost terminated computations at level k that are still to the left of c .

Given the above definition, let us see the complications concurrency poses to the maintenance of serial correctness. Now if we were to execute the above tree sequentially, the messages sent to higher level objects would be processed in the order given by the labels on the arrows. Note that this order can be derived by a

¹It is important to realize that even though the notions of serial correctness and serializability may appear to be analogous, they are not equivalent. Serializability theory in classical transaction management and concurrency control realms reasons about correctness and integrity in terms of the fundamental abstraction of a "transaction". Serial correctness on the other hand, is a more primitive notion as it does not recognize the abstraction or semantics of transactions, and is further more restrictive as it allows only a single serial order (i.e., the order of an RPC-based serial execution of computations (methods)). However, if we were to map individual computations to transactions and derive the transaction serialization order from the forkstamps, serial correctness amounts to a stricter form of the multiversion concurrency control notion of one-copy serializability [BHG87]. We intentionally do not give such a definition as this would give the impression that we are dealing with transactions, and would further introduce unnecessary formal machinery in our exposition.

²A computation b is said to be to the *right* of a computation a , if neither b nor a is an ancestor of the other, and b is encountered later than a in a depth-first traversal of the corresponding session tree, starting at the root. Similarly, a computation to the *left* will be encountered earlier in a depth-first traversal.

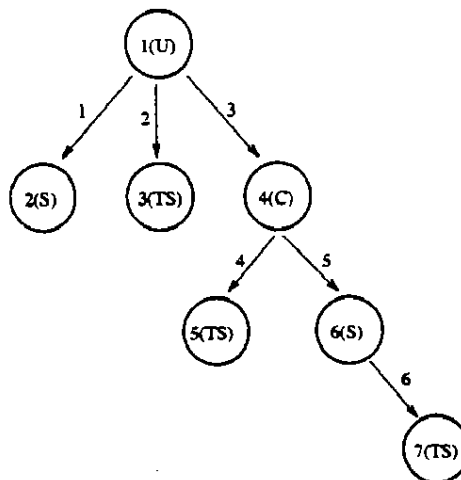


Figure 4.1: A tree of concurrent computations

left-to-right depth-first traversal of the tree. However, with concurrent execution it is possible that message managers 4(C) and 6(S) may terminate well ahead of 3(TS). Therefore our synchronization schemes must ensure that message manager 3 does not see any updates by message managers 4 and 6, since 4 and 6 are to the right of 3.

Solving the above synchronization problem using classical techniques, such as those based on locking and semaphores, is known to be insecure as they open up signaling channels. Also, it is not possible to implement these synchronization mechanisms in a kernelized architecture without introducing trusted subjects since we need the ability to write-down and read-up. Our solution instead relies on a multiversioning scheme. The scheme calls for multiple versions of objects accessed by a session to be kept in memory.³ Each version is uniquely identified with a timestamp, and can be thought of as a checkpoint in the overall progress of a tree of computations. Thus although 4(C) and 6(S) may terminate well ahead of 3(TS), we are guaranteed that a read-down request from 3(TS) will always read versions that existed before

³Note that there is no multiversioning on disk. Even if paging causes these versions to migrate to disk occasionally, they will not be visible to other sessions.

4(C) and 6(S) were started.

Given a computation, say c , the multiversioning scheme suggested above can provide synchronization when other computations to c 's right (in the tree) get ahead of c . But to guarantee serial correctness, we must in addition ensure that c itself does not get ahead of earlier forked computations to its left. For example, under a sequential execution of the tree of computations in figure 4.1, we would expect message manager 2(S) and its descendants (if any) to terminate before message manager 3(TS) to its right, is started. Message manager 3(TS) should thus see all the latest updates by 2(S) and any of its descendants. Allowing arbitrary concurrency may not ensure this. Thus, in addition to multiversion synchronization, we need to enforce some discipline on these concurrent computations by scheduling them in a manner that guarantees serial correctness.

A scheduling strategy which guarantees serial correctness must take into account the following considerations.

- The scheduling strategy itself must be secure in that it should not introduce any signaling channels.
- The amount of unnecessary delay a computation experiences before it is started should be reduced.

The first condition above requires that a low-level computation never be delayed waiting for the termination of another one at a higher or incomparable level.⁴ The second consideration admits a family of scheduling strategies offering varying degrees of performance. Some of these are discussed later in the next section.

⁴If this were allowed, a potential for a signaling channel is again opened up in a trusted subject architecture.

In summary, the maintenance of serial correctness requires careful consideration on how computations are scheduled as well as on how versions are assigned to process read-down requests. Collectively we have to guarantee the following constraints (as discussed in section 5.2, we assume that every computation is assigned a strictly increasing forkstamp that is consistent with the start order in a sequential execution):

Whenever a computation c is started at a level l ,

- **Correctness-constraint 1:** There cannot exist any earlier forked computation (i.e. with a smaller forkstamp) at level l , that is pending execution;
- **Correctness-constraint 2:** All current non-ancestral as well as future executions of computations that have forkstamps smaller than that of c , would have to be at levels higher or incomparable to l ;
- **Correctness-constraint 3:** For each level at or below l , the object versions read by c would have to be the latest ones created by computations such as k , that have the largest forkstamp that is still less than the forkstamp of c . If k is an ancestor of c , then the latest version given to c is the one that was created by k just before c was forked.

The above three constraints are sufficient to ensure serial correctness. We now state this formally as a theorem.

Theorem 4.1 *Correctness constraints 1, 2, and 3 are sufficient to guarantee serial correctness of concurrent computations in a user session.*

Proof:

Constraints 1 and 2 ensure that when computation c at level l is started, there will be

no more writes/updates forthcoming from earlier forked non-ancestral computations (the ancestral computations of c are those that are on the path from the root to c , in the computation tree). This guarantees that write operations by non-ancestral computations at levels l or below (and therefore inductively across all levels) will occur in the same relative order as in a sequential execution. Write operations from ancestral computations may, however, be issued in an order different from the sequential execution. Such out of order writes can affect the values obtained by later read operations from higher level methods. However, constraint 3 ensures that read down operations under concurrent execution will obtain the same state as in a sequential execution. To see this, consider any computation such as c at a level l . In a sequential execution all non-ancestral computations at lower levels and with smaller forkstamps than c , would have terminated before c . Thus higher level reads by computations such as c would obtain the last written versions by such non-ancestral computations. The ancestors of c on the other hand would be suspended in a sequential execution, waiting for c and its future children to terminate. Thus read operations issued by c should see the versions written by the ancestors just before they were suspended. Constraint 3 requires this and prevents c from reading out of order writes (versions) of its ancestors. \square

4.2 Maintaining Global Serial Fork Order

We now discuss an implementation consideration for our scheduling schemes which has to do with maintaining knowledge of the equivalent global serial order in which computations are forked within a user session. In scheduling various computations, such knowledge is used to determine when a computation will be started. In an architectural framework without multilevel trusted subjects, no single system component has a global view (such as the tree in figure 4.1) of the entire set of com-

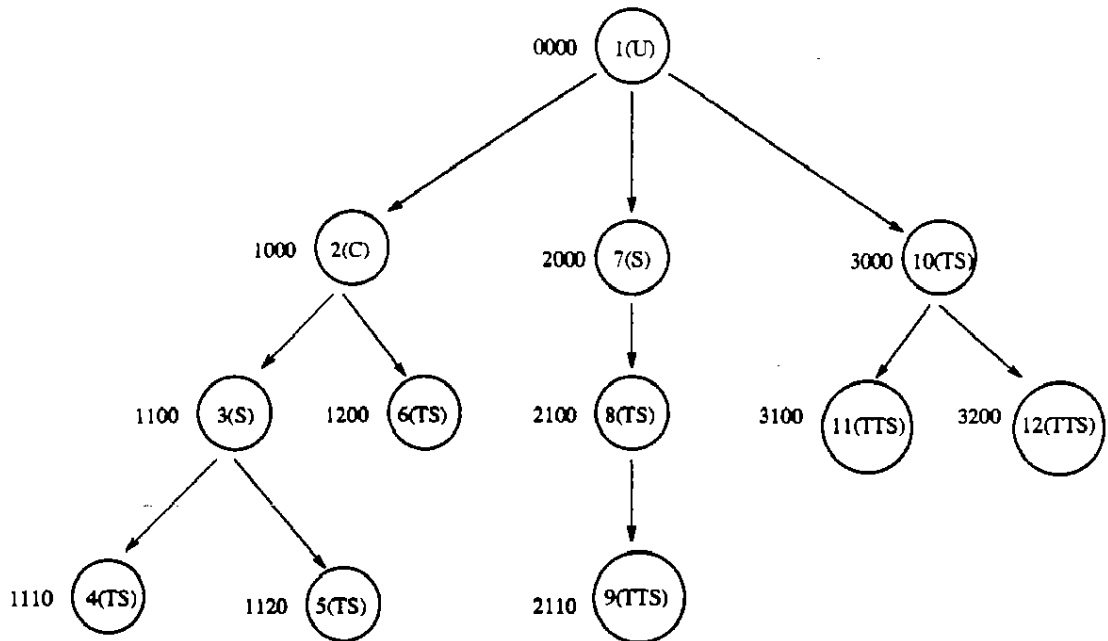


Figure 4.2: Generation of forkstamps for a session's computation tree

putations as they progress. In coordinating various computations, an individual level manager has to determine where in the global serial fork order, the computations at its level belong. One could be tempted to pursue a solution requiring the value of a global real-time clock to be appended to every message manager (computation) as it is forked. However, computations are not always forked in the equivalent serial order and thus a solution based on a real-time clock will not always work.

Consider now a hierarchical scheme to generate forkstamps that is independent of the scheduling strategy used. The forkstamps so generated, reflect the equivalent serial order of execution of the computations. Figure 4.2 shows a tree of computations and the forkstamps generated for it. Every message manager except the root, is assigned a unique forkstamp by the parent issuing the fork. The scheme starts by assigning an initial forkstamp of 0000 to the root message manager 1(U). Every subsequent and immediate child of the root is then given a forkstamp derived from this initial one by progressively incrementing the most significant (leftmost) digit by

one. To generalize this scheme for the entire tree, we require that with increasing depth along any path in the tree, a less significant digit be incremented. In general for a security lattice with a longest maximal chain of n elements, we need to reserve $p * (n - 1)$ digits for the forkstamp. In a lattice with l levels, and c compartments, $n = l + c$. The value of p would depend on the maximum degree of a node in a computation tree. For example if we assume that any computation sends a maximum of 99 messages to higher levels, then setting $p = 2$ would be sufficient. Even with large lattices and a high number of messages sent to higher levels, these numbers are reasonable.

We now show that the above forkstamping scheme captures the intended global serial order.

Theorem 4.2 *The hierarchical forkstamping scheme preserves global serial order.*

Proof:

We prove the above by contradiction. Consider any pair of computations c_1 and c_2 with forkstamps f_1 and f_2 respectively, such that c_1 was forked before c_2 (i.e., c_1 is to the left of c_2 in the computation tree). Assume $f_2 < f_1$. In other words, c_1 is forked earlier than c_2 but we assume that the former is given a later forkstamp. Let us see if this assumption can be contradicted.

By virtue of the fact that the forked computations form a tree, there will always be a common ancestor node a from which the paths to c_1 and c_2 fan out. Consider the edges from a to c_1 and c_2 as belonging to the paths p_1 and p_2 , respectively. Now consider the immediate children of a , say a_1 and a_2 that lie on paths p_1 and p_2 , respectively (for paths of length one this will be c_1 and c_2). Assume that we are using forkstamps with k digits denoted d_1, d_2, \dots, d_k , with d_1 being the most significant

digit. Of these k digits, let d_a be the digit assigned to (and incremented by) computation a , and d_{a/a_1} and d_{a/a_2} the values of the digit d_a for the forkstamps assigned to a_1 and a_2 , respectively. Now since the ancestor computation a forked c_1 before c_2 , it follows that the path p_1 will be to the left of path p_2 . Hence, our forkstamping algorithm will assign forkstamps to a_1 and a_2 derived from a 's forkstamp such that $d_{a/a_1} < d_{a/a_2}$. Now every descendant of a_1 (or a_2) on the path p_1 (or p_2) including c_1 (or c_2) will retain a_1 's (or a_2 's) value for the digits d_1, d_2, \dots, d_a . Hence it follows that the value of the digit d_{a/c_1} will always be less than d_{a/c_2} . Further, successive descendants of a_1 and a_2 in these paths will increment only digits $d_{a+1}, d_{a+2}, \dots, d_k$. These digits are less significant in place value than the digit d_a . Thus the value of the forkstamp f_1 of c_1 will be less than the forkstamp f_2 of c_2 . This contradicts the assumption that we stated out with, and hence the proof. \square .

An obvious implication of our forkstamping scheme is that only unique and acyclic forkstamps are generated. We conclude this section by stating this as a corollary to the above theorem.

Corollary 4.1 *The forkstamping scheme generates forkstamps that are unique and acyclic.*

Proof:

The proof follows from the following propositions:

1. Two computation nodes that lie on different paths originating from a common ancestor will have their forkstamps varying on the significant digit (position) that is incremented by the ancestor.
2. Nodes that lie on the same path will have forkstamps that increase with the length of the path.

3. The nodes form a tree and as such there no loops or parallel paths.
4. The forkstamps form a totally ordered set. \square

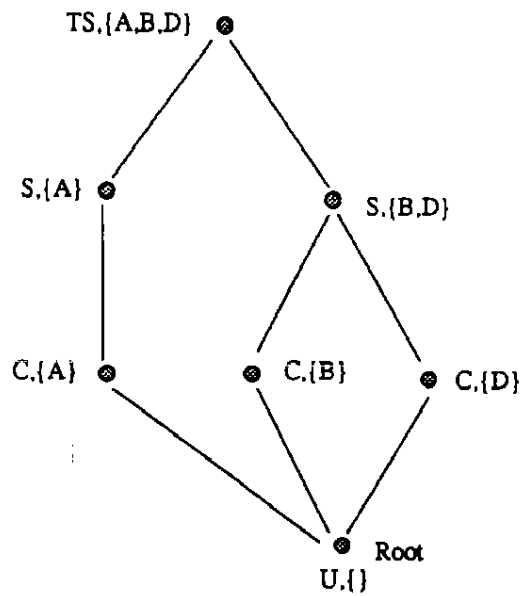
4.3 A Family of Scheduling Strategies

We now present a family of scheduling schemes. These schemes offer varying tradeoffs between performance and complexity when implemented under different architectures. We present two schemes, namely conservative and aggressive, that lie towards the ends of a spectrum of scheduling schemes. We also briefly mention a hybrid scheme for which the performance lies somewhere in between the above two. We also present a framework and metric for the comparative analysis of these schemes.

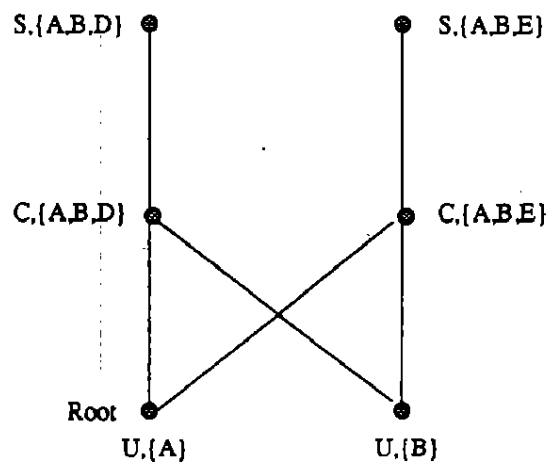
4.3.1 A Conservative Level-by-level Scheduling Scheme

Consider first a level-by-level scheduling scheme. We characterize this approach as being conservative, as opposed to being aggressive, since the objective here is not to maximize concurrency. In other words, a computation may be unnecessarily delayed before being started even if its earlier execution would not violate serial correctness. Although this scheme is not optimal in terms of performance, it does give insights into how concurrent computations can be scheduled and completed in a simple, yet secure, correct and distributed fashion. The conservative scheme maintains the following invariant:

Inv-conservative: A computation is executing at a level l only if all computations at lower levels, and all computations with smaller fork stamps at level l , have terminated.



(a) Hasse diagram for a lattice



(b) Hasse diagram for a partial order

Figure 4.3: Conservative level-by-level scheduling in lattice and partial order

Thus the basic idea is to execute forked computations in a bottom up fashion in the lattice, starting with the lowest level. At any point, only computations at incomparable levels can be concurrently executing. We thus begin with the root computation and allow it to run to completion. Meanwhile, all higher level computations that are forked by the root are unconditionally queued in forkstamp order at these higher levels, by the respective level managers. Upon termination of the root, its level manager signals that it is okay to release computations at all immediate higher levels by sending a WAKE-UP message to these levels. Thus when a level manager receives a WAKE-UP message from *all* immediate lower levels, it proceeds to dequeue and execute computations at its level one at a time in forkstamp order. Note that, at this point, this level manager is guaranteed that no more fork requests will be forthcoming from lower levels. Eventually, the level manager will find its queue to be empty. The next higher levels are then released through WAKE-UP messages.

For a more visual explanation of this level-by-level scheduling strategy, consider the lattice and partial order in figures 4.3(a) and 4.3(b). Consider the lattice first. On termination of the root computation at level $[U, \{\}]$, WAKE-UP messages are sent to all the immediate higher levels $[C, \{A\}]$, $[C, \{B\}]$, $[C, \{D\}]$, and queued computations at these levels are then released. Next, computations at $[S, \{B, D\}]$ are started when all those at the immediate lower levels $[C, \{B\}]$ and $[C, \{D\}]$ have terminated. Eventually, computations at the highest level $[TS, \{A, B, D\}]$ are started on the termination of computations at levels $[S, \{A\}]$ and $[S, \{B, D\}]$ followed by the receipt of a WAKE-UP message from each of these levels. Now consider the partial order in figure 4.3(b). When the root computation at level $[U, \{A\}]$ terminates, a WAKE-UP message is sent to all the immediate higher levels, namely $[C, \{A, B, D\}]$ and $[C, \{A, B, E\}]$ and computations at these levels may thus be running concurrently. Computations at level $[S, \{A, B, D\}]$ are released when a WAKE-UP message is received

from the only dominated class which is $[C, \{A, B, D\}]$. Similarly when computations at $[C, \{A, B, E\}]$ terminate, a WAKE-UP message is sent to level $[S, \{A, B, E\}]$ to release queued computations.

Figures 4.4(a) through 4.4(g) illustrate the progressive execution of the computation tree in figure 4.1, as governed by the level-by-level scheduling scheme. At each stage the termination of a computation results in the start-up of another. In this example, there can only be one computation executing at any given moment as the lattice is totally ordered. More generally, we could have multiple computations running, provided they are at incomparable levels. As shown in figure 4.4(a), the startup of the root computation has resulted in its forked children to be queued (the unborn computations have not yet been created, and are shown in the figures for visual completeness only). The subsequent termination of the root (see figure 4.4 (b)) has resulted in the forked child, at the lowest level 4(C), to be executed.

4.3.2 An Aggressive Scheduling Scheme

We now describe an aggressive scheduling algorithm. It is governed by the following invariant:

Inv-aggressive: *A computation is executing at a level l only if all non-ancestor computations, in the corresponding computation tree, with smaller fork stamps at levels l or lower, have terminated.*

We characterize this as an “aggressive” scheme as every attempt is made to execute a forked computation immediately. The above invariant implies that if a computation is denied immediate execution, then there must be at least one non-ancestral lower level computation with an earlier forkstamp, that has not terminated. The invariant ensures that the correctness constraints 1 and 2 are never violated. The correctness

of read-down operations is again dependent on multi-versioning.

The major differences between the aggressive and conservative schemes can be summarized as follows:

- On being forked, a computation may be immediately started, if doing so would not violate the invariant **inv-aggressive**.
- The termination of a computation may result in the start-up of the next queued computation at the same level as well as multiple computations at other higher levels.
- A wake-up is sent to a higher level only if there exists at least one queued computation pending execution at the higher level.
- A level may receive multiple wake-up messages before all its queued computations are released.

Figure 4.5 illustrates how a tree of computations can advance to termination under the aggressive scheme. In particular, we note that the termination of a computation may result in multiple start-ups of others at higher levels, even with a totally ordered security lattice, so long as the invariant is not violated (see figure 4.5(c) where computations 3(TS) and 6(S) are started on termination of 2(S)). We also observe that with aggressive scheduling, by the time the first four terminations have occurred, namely, 1(U), 2(S), 3(TS), and 5(TS), the entire tree of computations has been released for execution (see figure 4.5(e)). Now compare the progress of this tree under conservative scheduling where the first four terminations as shown in figure 4.4 (e), still leaves four others queued and awaiting execution. In summary, the tree progresses to termination at a much faster rate, under the aggressive scheduling scheme.

4.3.3 Hybrid Schemes

We now consider a variant of the level-by-level scheduling scheme. It is a hybrid scheme as it combines both conservative and aggressive approaches. The basic idea is simple. As in the conservative level-by-level scheme, we execute computations on a level-by-level basis. However, when a computation is allowed to be active (by virtue of its level), we allow its immediate children to execute as well, if doing so would not violate serial correctness. Figure 4.6 illustrates how a tree of concurrent computations advances to completion under this hybrid scheme.

To get a quick comparison of the conservative, hybrid, and aggressive schemes, consider the trees in figures 4.4(e), 4.6(e), and 4.5(e), each with four terminated computations. In the conservative scheme, this leaves two computations still pending, while in the hybrid scheme we have only one computation pending execution. On the other hand, with the aggressive scheme, the termination of four computations leaves no computations pending start-up.

4.3.4 A Framework and Metric for Comparative Analysis

The conservative and aggressive schemes discussed above can be seen as two that approach the ends of a spectrum of secure and correct scheduling strategies. This is because it is meaningless to come up with any algorithm that does worse than the conservative one, in terms of the degree of concurrency allowed. At any given time, if there is a computation active at a maximal level in the lattice, then no other computations may be concurrently active. The conservative scheme thus exhibits the least meaningful degree of concurrency within a session. The only way to do worse would be to allow computations at incomparable levels in the lattice to execute one at a time (and not to mention the fact that this would be insecure due to sideways signaling channels). On the other hand with the aggressive scheme, we can poten-

tially have concurrent computations running at every level. This can happen if a computation is forked at the highest level in the lattice, and this is followed by consecutive fork requests where each request is at the next lower level and the lifetimes of these computations are long enough to overlap. One can always increase the degree of concurrency by exploiting intra-level concurrency. But conflicts at the same level can be easily handled by well-known concurrency control techniques. We do not explore this issue further in this thesis as it lies outside the scope of the execution model and scheduling protocols we present.

We now develop the notion of *delay-degree* as a metric for analyzing scheduling strategies. We demonstrate how by varying this metric, we can derive and admit a family of scheduling strategies offering varying degrees of concurrency, while guaranteeing confidentiality and serial correctness.

We begin with some definitions.

Definition 4.2 *A level is inactive if no computation is executing at the level.*

Definition 4.3 *A level is active if there exists an executing computation at the level.*

Definition 4.4 *We say a level l is serial-execution enabled (or s-enabled for short), if there exists at least one forked computation c , at l , and there are no active or queued non-parent computations with smaller fork stamps than c , at level l or below.*

Intuitively, when a level is s-enabled, executing the next computation at the head of the queue at this level will not violate serial correctness. A computation that is denied execution by a scheduling scheme when its level becomes s-enabled is therefore experiencing an unnecessary delay. We build on this observation and extend it below to an entire security lattice in order to formulate a metric for analysis purpose.

Definition 4.5 *A scheduling algorithm introduces an unnecessary delay whenever any level is s-enabled but remains inactive.*

Definition 4.6 *We say a chain of n security levels in a lattice is fully-enabled whenever every level in the chain is concurrently s-enabled.*

Definition 4.7 *We define a computation tree to be a full-enabler for a given security lattice, if it causes a longest maximal chain in the lattice to be fully-enabled.*

Thus when a maximal chain in the lattice is fully-enabled, computations can be concurrently running at every level in the chain. However, when scheduling is governed by some scheme, it is only certain scenarios that can cause such chains to be fully-enabled. We characterize below the computation trees associated with such scenarios as realizers.

Definition 4.8 *For a given scheduling algorithm and security lattice, we define a realizable full-enabler (or realizer for short) to be a full-enabler, which when scheduled by the algorithm, causes a longest maximal chain in the lattice to be fully-enabled.⁵*

Definition 4.9 *We say a realizer has a delay-degree (d-degree) of k for some scheduling algorithm, if it causes k computations to experience unnecessary delays.*

Definition 4.10 *Given a security lattice (SC), a scheduling algorithm (A) is considered to have a delay-degree (d-degree) of k , where $k = \max \{d\text{-degree of all realizers for SC under A}\}$.*

⁵It is important to note that our framework is not restricted to lattices; it applies equally well to maximal chains of partial orders.

Given a set of secure scheduling schemes, we can now use their d-degrees as a basis for comparison. We thus need to derive the d-degree for any given scheduling scheme. To do this, we consider all the realizable full-enablers (realizers) and observe the maximum number of computations, excepting the root, that are denied immediate execution, on being forked. This number would give us the d-degree.

As an illustration, consider the full-enabler trees in figure 4.7 for a lattice with a longest maximal chain of three levels U, C, and S (where $U < C < S$). For the aggressive scheme, we see that both trees are realizers and in either cases no computation would be unnecessarily delayed. For the conservative scheme, only the tree in 4.7(a) is a realizer and we see that computations 2(S) and 3(C) would be unnecessarily delayed. For a further illustration, consider all the full-enabler trees for four levels U, C, S, and TS, as shown in figures 4.8(a) through 4.8(e). All the trees are realizers for the aggressive scheme, and in each case no computation would be unnecessarily delayed. However, only the tree in figure 4.8(a) is a realizer for the conservative scheme and the computations 2(TS), 3(S), and 4(C) would be unnecessarily delayed.

In both of the examples above, we see that the aggressive scheduling scheme would have a d-degree of zero (0), while the conservative scheme would have a d-degree of $n - 1$ for a lattice with a longest maximal chain of n elements. These results are general and not specific to these two examples. To be more precise, the d-degree, say k , of a scheduling scheme holds true for any lattice with a longest maximal chain of n elements, as long as $k \leq n$. Also, it follows that for any scheduling scheme with a d-degree of 0, a level is inactive only if it is not s-enabled.

Now are there other scheduling schemes that have d-degrees between the extreme values of 0 and $n - 1$? To answer this question, let us look at the hybrid (variant of the level-by-level) scheduling scheme discussed earlier. Recall that with the level-by-level scheme, computations are executed one level at a time. Thus at

any given time, there is a *current-level* at which computations are dequeued and executed. While our variant would also require that computations be dequeued and executed one level at a time, it would in addition permit the execution of all the immediate child computations of any active computation at the current-level. To derive the d-degree of this variant, consider again the full-enabler trees in figures 4.7 and 4.8. Both trees in figures 4.7(a) and 4.7(b) are realizers with d-degrees of 0 and $n - 2$ respectively, and thus giving a d-degree of $n - 2$ for this variant (i.e., $\max\{0, n - 2\}$). In figure 4.8 the trees (a), (c), (d), and (e) are realizers with d-degrees 0, $n - 2$, $n - 3$, and $n - 3$ respectively, giving again a d-degree of $n - 2$ for this variant. It thus introduces fewer delays, due to increased concurrency, than the conservative scheme with a d-degree of $n - 1$. We conjecture that by varying the metric d-degree, one could derive several scheduling schemes.

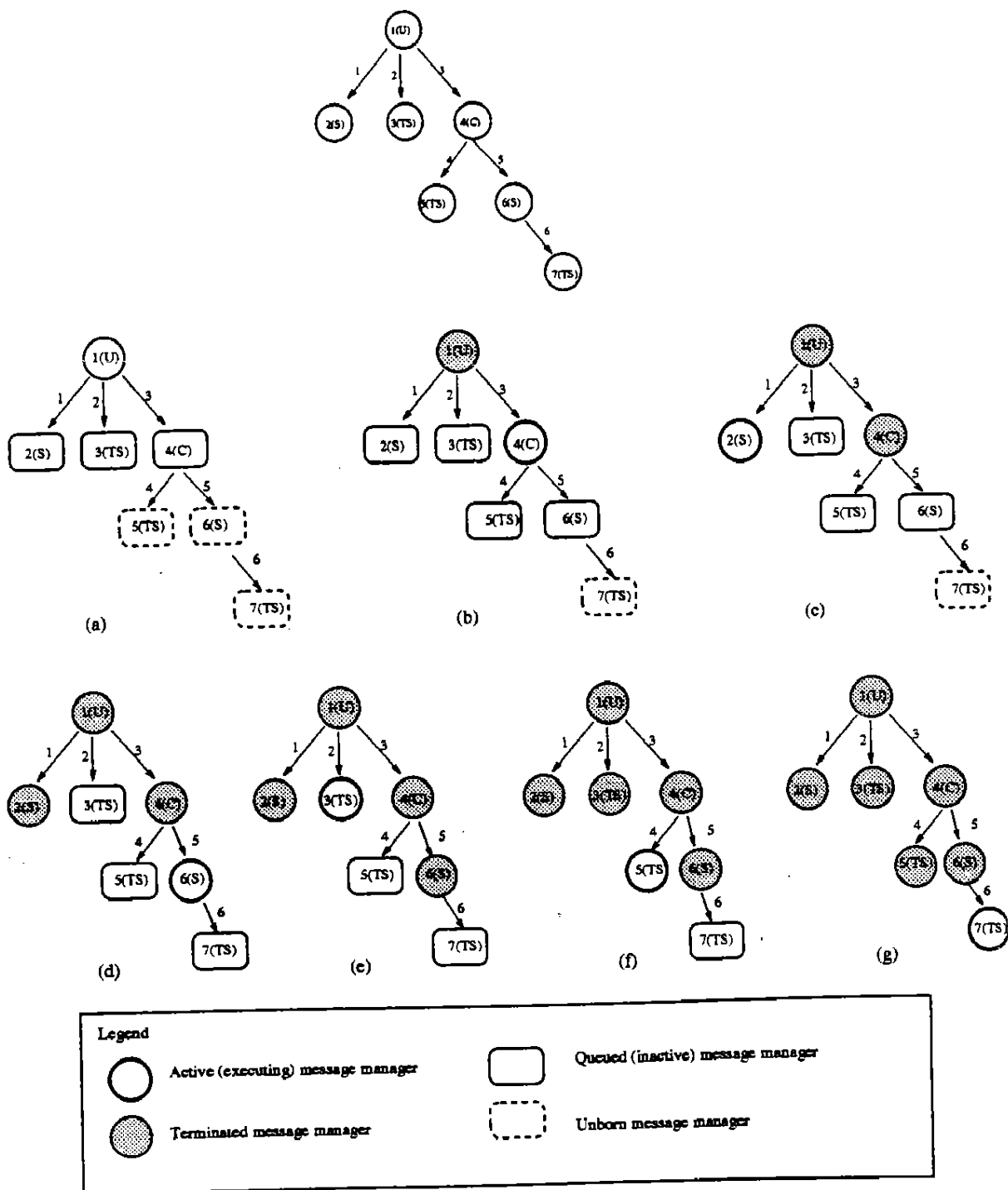


Figure 4.4: Progressive execution under conservative scheduling

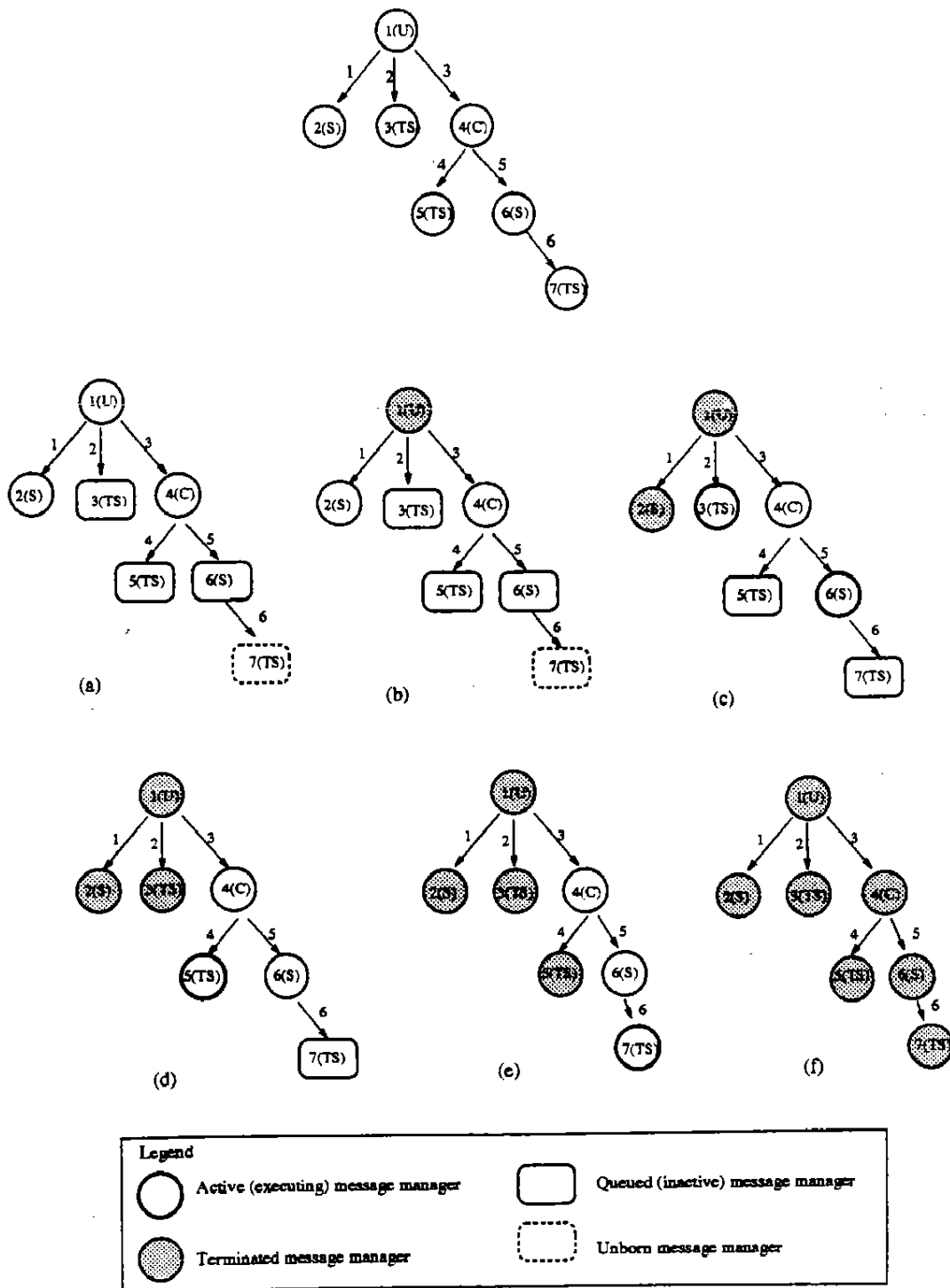


Figure 4.5: Progressive execution under aggressive scheduling

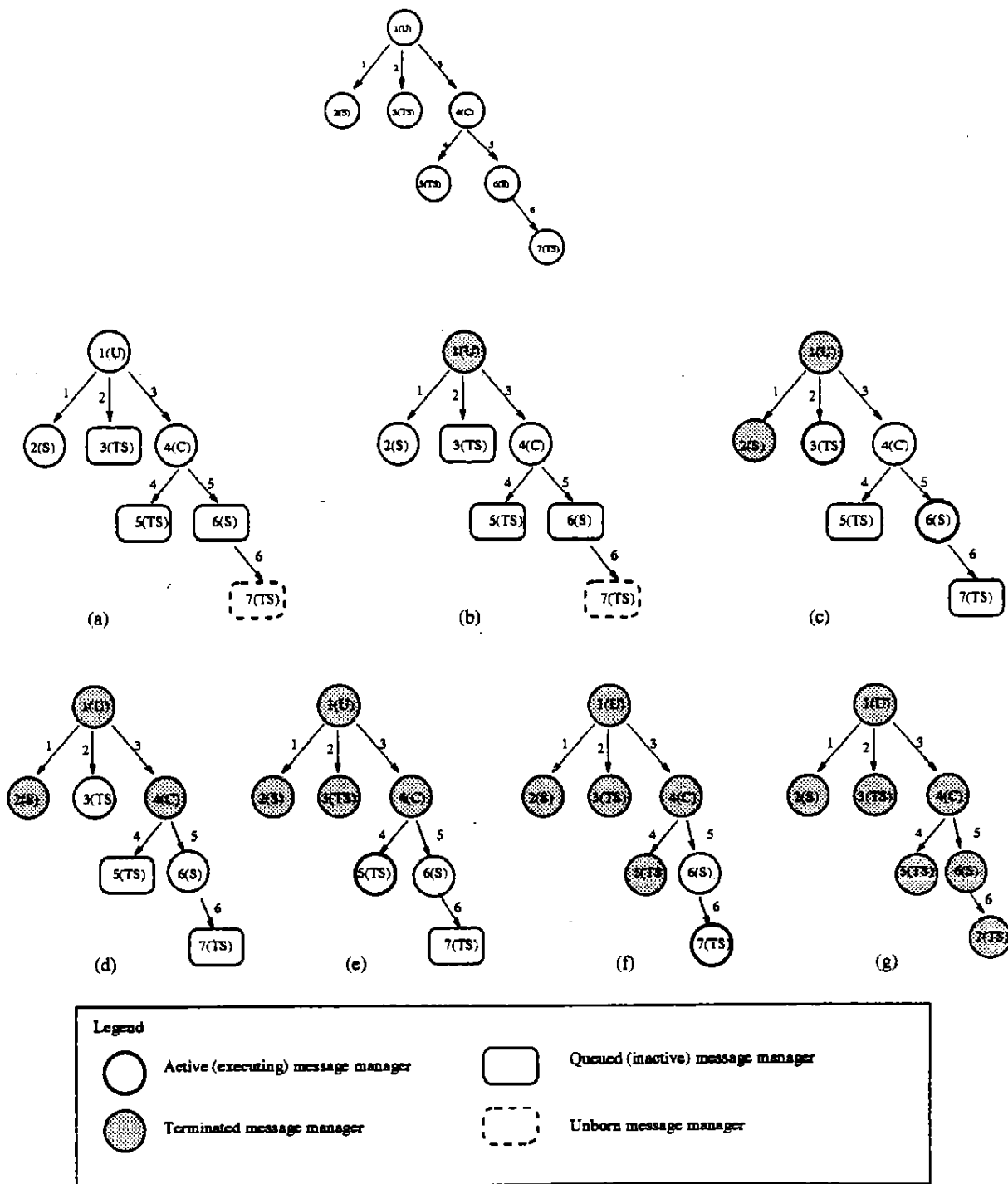


Figure 4.6: Progressive execution under hybrid scheduling

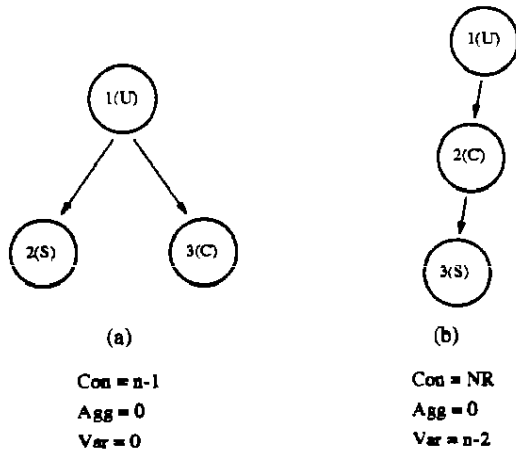


Figure 4.7: Full-enablers for a longest maximal chain of 3 elements ($n = 3$)

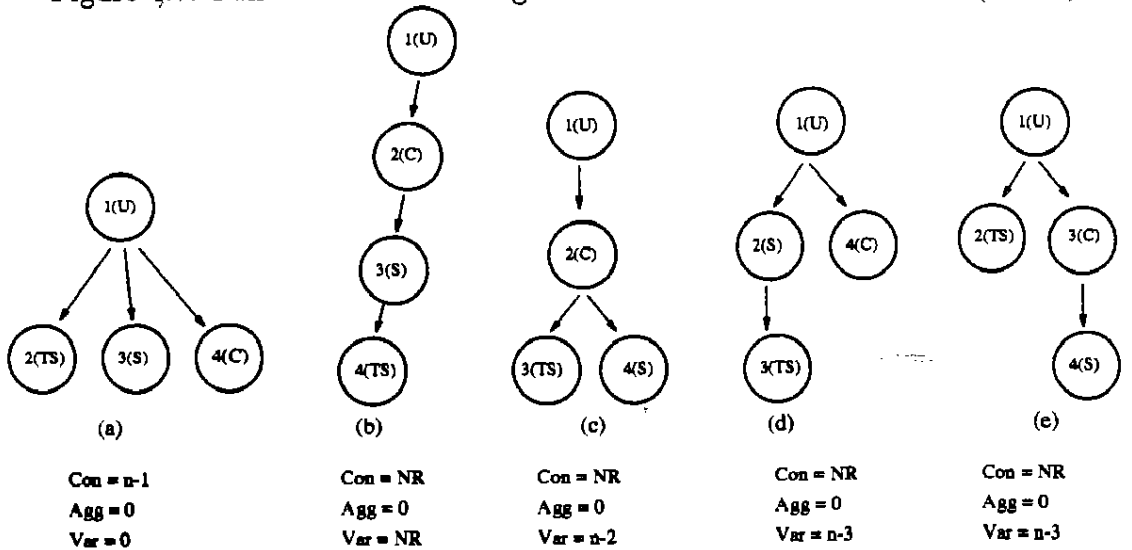


Figure 4.8: Full-enablers for a longest maximal chain of 4 elements ($n = 4$)

Chapter 5

Trusted Subject Architecture Implementation

In this chapter we elaborate on implementing the message filter model and the aggressive scheduling scheme under a trusted subject architecture. We begin by discussing a trusted subject architecture for multilevel object-based systems. This is followed by the description of some data structures and various scheduling algorithms. We finally present some theorems and their associated proofs to demonstrate that these scheduling algorithms preserve integrity and confidentiality.

5.1 Architecture

Figure 5.1 illustrates the trusted subject architecture. As mentioned before, the session manager is the trusted subject in this architecture. It is thus a multilevel process and coordinates single-level (untrusted) message manager processes. The session manager is a long-lived process that is created when a session starts and deleted only when the session eventually terminates. A session manager may create several short-lived message-manager processes. Whenever a write-up message is issued, a message manager process is created to service the request, and it implements the message filtering functions.

The interface between a message manager and its local session manager consists of fork, terminate, and start calls. A fork is issued by a message manager to request creation of a new message manager. A terminate call is issued by a message

manager to its session manager and signals termination. A start call is issued by a session manager to a message manager to initiate the execution of the message manager.

What is the motivation, if any, for the trusted subject architecture and implementation? The main advantage is the simplicity with which the scheduling algorithms can be implemented due to the availability of a trusted subject. A session manager always maintains a global snapshot of a session's tree of computations as they progress. With the help of such a global snapshot (view), it is able to coordinate the various concurrent and implement the scheduling algorithms. As we will see in the next chapter, without such a global snapshot, the coordination of concurrent computations has to be achieved in a distributed fashion and this complicates the implementation of the scheduling algorithms.

The above advantage of using a trusted subject for scheduling does come at a price. We now have to provide assurance that such a trusted subject cannot leak information. We later give a noninterference argument to demonstrate that the session manager cannot leak information while coordinating various scheduling strategies and being exempt from mandatory access control rules.

5.2 Scheduling Algorithms

Recall that the session manager in the trusted subject architecture always has a global snapshot of the tree of concurrent computations as they progress to termination. The availability of such a global snapshot significantly reduces the complexity of implementing scheduling algorithms. In fact, we observe that the implementation of the aggressive scheduling is no more complex than the implementation of the conservative one, and as such the latter provides no significant advantage. Hence, for brevity we discuss here only the aggressive scheme for the trusted subject architecture.

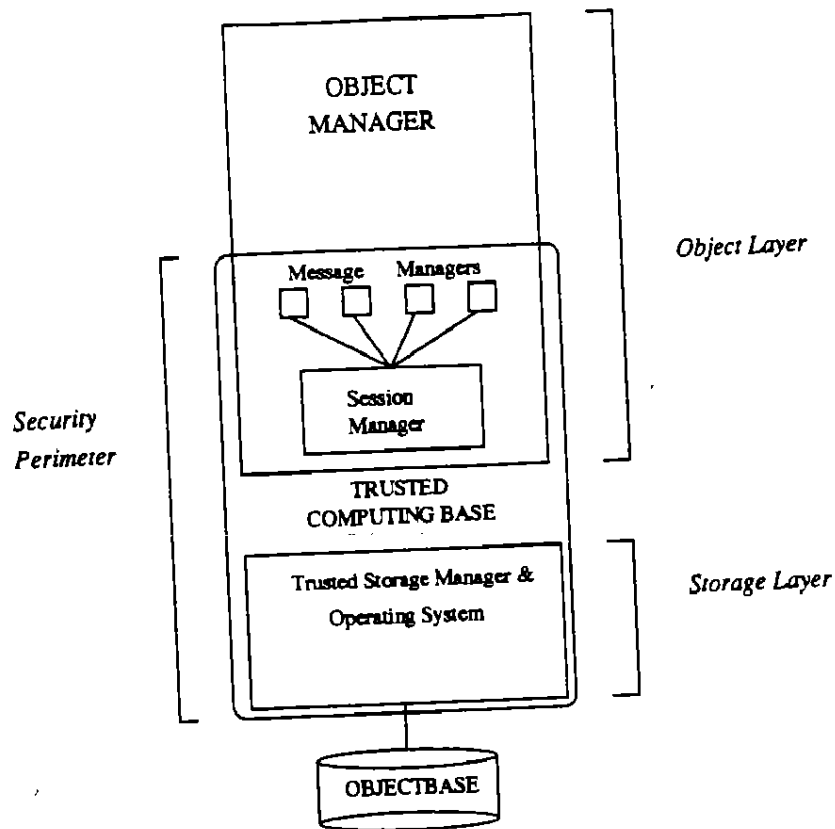


Figure 5.1: A trusted subject architecture with TCB

Before discussing the algorithms in detail, we describe the data structures used by the session manager. Recall from chapter 4 that our approach to synchronizing concurrent computations was based on a multiversioning. Every version of an object when created, is assigned a unique timestamp. The session manager maintains the following data structure to keep track of initial versions.

- **Init-stamp:** This is a global table of timestamps with one entry per level. It identifies the initial version of objects at every level that exists before a session starts. An individual message manager can see that portion of the table which is for levels dominated by that message manager.

The session manager also maintains a tree structure that reflects the progress of the

concurrent message managers forked in a session. Every forked message manager is represented by a node in the tree that contains the following information attributes:

status: this can be one of the following: active, terminated, queued;
level: the level of the message manager;
local-stamp: a local table of timestamp entries with an entry at each level dominated by the message manager and identifying the versions at the level that will be used to process read-down requests;
forkstamp: forkstamp issued by the parent message manager;
parent: pointer to parent message manager;
wstamp: This is timestamp entry indicating the next version that will be written by the message manager;
object: receiving object;
message: message;
p: message parameters;

A message manager's **local-stamp** vector is initialized in two phases, with the first one undertaken when a message manager is forked and the second one deferred until the message manager actually starts. For a message manager just forked, the first phase entries identify the versions to be read at the levels of ancestors, on the path from the root to itself (i.e., the path in a computation tree for a session). These first phase entries are actually obtained by a message manager from another vector that is passed along by its parent. Such a vector can be seen as one that is incrementally constructed along a path in the computation tree. To do this, every message manager is required to save the timestamps in the vector (**astamps**) obtained from its parent and on issuing a **fork**, to reconstruct a new vector to give to its child (see figure 3.5). This newly constructed vector will contain the timestamps from the old vector appended with the write stamp **wstamp** at the level of the issuing message manager. Finally, in the second phase we obtain **local-stamp** entries for the levels that did not participate in phase one (this is done in the **start-trusted-agg** procedure of figure 5.3).

```
Procedure fork-trusted-agg(level-parent, level-create, forkstamp, astamps)
{
  Let parent be the node issuing the fork;
  Let child be a new message manager node;

  Make child the rightmost child of parent;
  child.level ← lub[parent.level, L(O2)];

  child.forkstamp ← forkstamp;

  %Begin phase 1 of acquiring local-stamp entries
  For (every level l ≤ level-parent)
  do
    initialize child.local-stamp table entries from astamps;
  End-For

  If in a depth-first traversal of the tree starting at the leftmost path and
  until child is traversed, there exists a non-ancestor node, say n, with
  {n.level ≤ child.level and n.status = active or queued}
  then child.status ← queued;
  else
    start-trusted-agg(child);
  end-if
}
end procedure fork-trusted-agg;
```

Figure 5.2: Session manager algorithm for FORK

```
Procedure start-trusted-agg(nn)
{
  %Let node nn represent the message manager to be started

  % Complete phase 2 of acquiring local-stamp entries
  %Update timestamps from terminated message managers to the left
  Initiate a depth-first search of the tree until node nn is traversed such that:

  If the level l of a node n traversed is not a level of any of the ancestors of
  nn
  and  $l < nn.level$ 
  then
     $nn.local-stamp[l] \leftarrow n.wstamp;$ 
  end-if

  %Update remaining local timestamp entries from the Init-stamp table
  If there exists a level l lower than the level of nn and which is neither
  the level of a node traversed in the tree nor of an ancestor of nn
  then
     $nn.local-stamp[l] \leftarrow Init-stamp[l];$ 
  end-if

  execute(nn);
}
end procedure start-trusted-agg;
```

Figure 5.3: Session manager algorithm for START

```

Procedure terminate-trusted-agg(lmsgmgr, wstamp, forkstamp)
{
  Let term be the node that terminated at level lmsgmgr;
  % Mark this node as terminated
  term.status ← terminated;

  % See if any queued nodes can be started
  Initiate a depth-first traversal of the session tree such that:

  If for every leaf node say leaf, that is traversed to the right
  of term such that leaf.level ≥ lmsgmgr, there exists
  no previously traversed non-ancestor node p with {p.level ≤ leaf.level and
  p.status = active or queued}
  then
    start-trusted-agg(leaf);
  end-if
}
end procedure terminate-trusted-agg;

```

Figure 5.4: Session manager algorithm for TERMINATE

A high-level pseudocode specification of the session manager algorithms to implement the aggressive scheduling scheme is shown in figures 5.2, 5.3, and 5.4. The algorithms make extensive use of the tree structure representing the various message managers. Let us discuss these algorithms in more detail. They are basically designed to ensure that the invariant *inv-aggressive* presented in chapter 4, is never violated. For easy reference, we give the invariant below:

Inv-aggressive: *A computation is executing at a level l only if all non-ancestor computations, in the corresponding computation tree, with smaller fork stamps at levels l or lower, have terminated.*

Whenever a fork request is received (see the procedure in figure 5.2), the session manager updates its tree structure by creating a node for the forked message manager and making it the right most child of the parent node issuing the fork. The procedure then records the forkstamp for the newly forked message manager that

has been passed on by the parent (i.e., the message manager that generated the fork request). This is followed by the first phase of the initialization of the local-stamp entries. The session manager then checks to see if the forked node can be started immediately. To do so, a depth first traversal of the tree is made starting at the leftmost path until the newly inserted leaf node is reached. If during this traversal we find another node, active or queued, at the same or a lower level, the newly inserted node is queued and thus forced to wait.

The processing of a terminate request begins by updating the status of the node to terminated (as shown in figure 5.4). We then check to see if this termination can release other nodes queued up. In determining this, our invariant leads to the property that any nodes started as a result of a termination have to be to the right of the terminated node and at a equal or higher level (and of course, these nodes have to be leaves in the tree). Thus a depth first traversal of the tree is once again initiated. Now as in the fork case, a leaf node is allowed to execute if and only if required by the invariant. It is important to note that a termination may result in more than one node being started. For example in figure 4.5(c) the termination of message manager node 2 (secret) results in nodes 3 (top secret) and 6 (secret) being started.

Both the Fork and Terminate algorithms utilize a common Start procedure (shown in figure 5.3) by which message managers are started. This procedure is primarily concerned with the completing the update of the local-stamp table entries of the node to be started. Recall from our previous discussion that the first phase of updating the local-stamp entries is achieved at fork time. The second phase is now accomplished from the following sources.

1. **Terminated left nodes:** For levels dominated by a node's level, and for which timestamps were not obtained from the ancestors, the start algorithm looks to

the subtree of computations to the left of the node to be started. The timestamp of the last written versions at such levels is obtained from the last forked message manager (or rightmost node to the left of the node to be started) which wrote at these levels.

2. **Init-stamp table:** If there are levels for which timestamps could not be obtained from phase 1 or from terminated left nodes, the algorithm then retrieves the timestamps from the global Init-stamp table maintained by the session manager. This is because objects at these levels have not been updated so far in the session. Thus the initial versions of objects that existed before the session started at these levels should be used by the starting message manager. The timestamps in the Init-stamp table identify such versions.

Once all the local-stamp entries have been collected, the message manager is started (executed). Thus once a message manager starts, its node in the tree will have all the timestamps necessary to process read down requests for objects classified below its level. These timestamps are never modified in the local-stamp table after start up. However, the timestamp entry stored in the variable *wstamp* is dealt with differently. On start, the timestamp is incremented unconditionally before the first write (update) operation and subsequently incremented after every fork request issued to the session manager. Thus the timestamp passed on to the forked children by a message manager will vary. Each value identifies the state of the objects at the level of the message manager as of the time the fork was issued.

Proof of correctness.

We now state and prove that the aggressive scheduling algorithms under the trusted subject architecture preserve the invariant *inv-aggressive*.

Theorem 5.1 *The aggressive scheduling algorithms for the trusted subject architecture maintain the invariant **inv-aggressive**.*

Proof:

Message managers get started only in the body of the **fork-trusted-agg** and **terminate-trusted-agg** procedures, with a call to the procedure **start-trusted-agg**. In the procedure **fork-trusted-agg** in figure 5.2, the precondition to executing the statement **start-trusted-agg** when the node *child* is forked is that there exist no non-terminated node to the left of *child*, at levels dominated by *child*. This condition continues to hold after *child* is started. Whenever this condition holds, the invariant is maintained. A similar precondition holds before after the execution of the statement **start-trusted-agg** in procedure **terminate-trusted-agg** of figure 5.4 for every leaf node traversed to the right of the just terminated node in the tree. Thus the invariant **inv-aggressive** is maintained. \square

We now state and show how the invariant **inv-aggressive** maintains serial correctness under our implementation. We state this as a corollary.

Corollary 5.1 *The aggressive scheduling implementation maintains serial correctness.*

Proof:

We basically have to show how the correctness constraints 1, 2, and 3 are maintained. For a computation to be dequeued and successfully started, invariant **inv-aggressive** requires all earlier forked computations at level *l* or lower, to have terminated. But this is what is precisely required to maintain correctness constraints 1 and 2. The argument for the maintenance correctness constraint 3 is independent of the scheduling algorithm used. The **local-stamp** table entries collected in the first phase at fork time by *child* reflect versions identifying the states of objects written at the level of

these ancestors before each successive child in the ancestral path was forked (see procedure **fork-trusted-agg** in figure 5.2). The second phase entries on the other hand identify latest versions written at lower levels for which there were no ancestors. In summary, the read down operations when assigned versions identified by the **local-stamp** entries, will read the same object states as in a sequential execution, thus ensuring that correctness constraint 3 is maintained. Thus all the three constraints are maintained and serial correctness follows. \square

Proof of termination

We now prove that the aggressive scheduling scheme terminates. In order to proceed with a proof of termination, we assume that once a method (computation) is started, it runs uninterrupted to completion. Obviously, such an assumption can be valid only if the body of the method contains no errors such as an infinite loop. We assume that there is some time-out mechanism in place, to handle such situations. We argue termination of individual computations by formally stating and proving the lemma below:

Lemma 5.1 *Once a computation is started, it is guaranteed to terminate.*

Proof: The proof follows from two observations:

1. Whenever a computation issues a **send** which results in a FORK, it is not blocked, but rather runs concurrently with the receiver computation. Now if a computation only issued forked new computations, it is guaranteed to run to completion and terminate since only a finite number of FORK requests can be issued.
2. Whenever a method issues a **send** that does not result in a FORK, it will be blocked and in general this could result in a chain of blocked methods. However,

there will always be a method executing and progressing to termination at the end of such a chain, and if there are no cyclical send relationships, such a method will eventually resume its blocked predecessor. It follows that any blocked method will be resumed eventually and allowed to run to completion in finite steps. \square

We now state and prove formally as a theorem that a session will terminate.

Theorem 5.2 *The aggressive scheduling algorithms for the trusted subject architecture guarantee termination of user sessions.*

Proof:

A message manager (computation) m can be denied immediate execution only at the moment it is forked. If this happens there has to be at least one non-terminated computation to the left of m in the tree. Now by lemma 5.1 every running computation will eventually have to terminate. Also, we know that every termination causes at least one other queued computation to be started. Thus in finite steps all computations to the left of m will terminate, causing m to be started. We can apply this argument to every queued computation and it follows that the entire session will terminate. \square .

5.3 Proof of Confidentiality

We now give a confidentiality proof for the trusted subject architecture and implementation. Recall that in this architecture, the session manager is a trusted multilevel process that accepts inputs from different security levels, processes them, and outputs information at various levels. Now since the session manager is the only trusted process in this architecture, it follows that any confidentiality leaks would have to be introduced and traceable to the session manager.

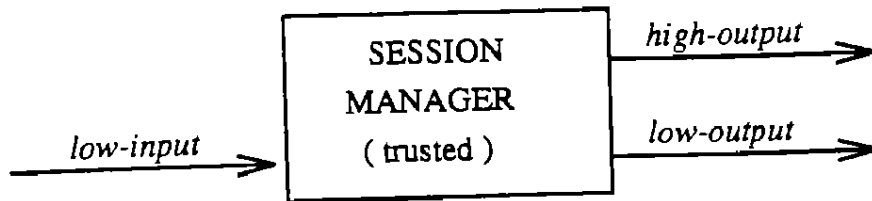


Figure 5.5: The two-step processing cycle of the session manager

Before developing a formal security proof, we first give an intuitive argument for the confidentiality of the session manager. We view the session manager as a black box accepting inputs and producing outputs. More precisely, the inputs are the **fork** and **terminate** requests issued by message managers at various security levels and the outputs are the **start** requests issued by the session manager requesting the start-up of previously forked (and perhaps queued) message managers. We assume that these inputs are the sole means by which information enters the process and the outputs are the sole means by which information leaves the process.

Consider first just two security levels low and high ($low < high$). A fork request issued by a message manager (computation) at low will form a low-input (low-fork) to the session manager. Now such an input can generate only a high output (high-start), when the fork request is processed. Now consider three levels low, med, and high. The termination of a computation at level med can result only in the start-up of another computation at level med or high. Thus in either case, an input generated at level l and received by the session manager, can produce an output only at levels l or higher. Now if we examine the processing steps of the session manager, we see that it is a repetitive two-step cycle of accepting inputs at a level l and producing the associated outputs at l or higher (as shown in figure 5.5). Thus in general any information flow through the session manager occurs only in an upwards direction in a lattice.

Having given an intuitive argument for confidentiality by demonstrating that information flow is always upwards in security levels, we now develop a more formal and rigorous proof using the theory of non-interference [GM82, GM84]. We consider the reception of inputs as well as the generation of outputs to be discrete events. As in [McC90] we call the events less than or equal to a level l as belonging to the the *view* of that level, and all other events as *hidden* from l . The basic idea of noninterference can be stated as follows: A subject s_1 is said to be noninterfering with subject s_2 if no action issued by s_1 can influence the future output of the system to s_2 . An obvious approach to establishing noninterference is to purge all hidden inputs and demonstrate that the events observed in the view for a lower level subject remains unchanged.

We begin with some definitions and formalisms.

Definition 5.1 *We define an event as a triple $(type, l, tstamp)$ where $type \in \{fork, term, start\}$, l is the level of the message manager (computation) from which the input originated or the level of the message manager to which an output is directed, and $tstamp$ is a timestamp indicating the elapsed time since the occurrence of the last event at l .*

Definition 5.2 *Given any security level, l , we define the events at less than or equal to l as belonging to a set called the view of l and all other events as belonging to a set called hidden from l .*

Definition 5.3 *Given any security level, l , we define the subset of events in the view of l that are at levels strictly below l as belonging to the set lower-view.*

Definition 5.4 *Given two event sequences β_1 and β_2 , we say that they are l -equivalent (denoted as $\beta_1 \approx_l \beta_2$) if they contain the same events, in the same relative order, for*

levels l and lower (i.e., they contain the same values for the event triples and the events associated with these triples appear in the same relative order in both sequences).

Definition 5.5 Given a sequence β , we define a purge function $\text{purge}(\beta, l)$ as one that returns the sequence β but with all events of the form $(\text{type}, l_e, \text{tstamp})$ removed (purged) whenever $l \not\geq l_e$.

Given any input sequence α_1 , which when processed by the session manager produces an output sequence β_1 (denoted $\alpha_1 \rightarrow \beta_1$), noninterference requires us to show the following: If for every level l , $\text{purge}(\alpha_1, l) \rightarrow \beta_2$, then $\beta_1 \approx_l \beta_2$.

Before proceeding on a formal proof that the session manager is noninterfering, we list our assumptions:

- **Input-totality.** If we view the session manager as a state machine, this assumption states that the session manager (or state machine) can accept inputs in any state. This ensures that the session manager is not conveying any information by accepting inputs.
- **Input-output atomicity.** This assumption requires the session manager to accept an input and produce the corresponding outputs, if any, atomically. In other words, in the interval between the acceptance of an input and the subsequent processing and generation of the corresponding outputs, the session manager cannot be interrupted, especially by other inputs.

The assumptions of input totality and input-output atomicity may seem at first to be irreconcilable. After all, if the session manager cannot be interrupted in the interval between the acceptance of an input and the production of the corresponding output, how would it be capable of accepting other inputs that come within such an

interval? Assume for a moment that inputs arrive at the session manager boundary synchronized with clock ticks that are a constant interval apart. It is important to note that such an interval can be chosen to deal with worst case arrival rates of inputs. The session manager is required to accept an input at a clock tick, and produce all the corresponding outputs, before the next clock tick. In other words, at every clock tick, the session manager is ready to accept an input, and within clock ticks cannot be interrupted to accept other inputs.

In the above model, given an input, we require that the corresponding outputs be produced within the same interval. In other words, the outputs cannot spill over to time intervals between subsequent clock ticks. However, one needs to approach the implementation of this requirement with caution. In particular, the timing of the outputs within an interval should not be used to build a channel. Hence we require the scheduler to hold off all outputs until the expiration of the interval. Upon expiration, the outputs are delivered as a batch to a lower level subsystem or operating system.

The realization of the input-output atomicity assumption also requires that the tree data structure implementation utilized by the session manager be an "ideal" one. By ideal we mean that the elementary data structure operations such as the insertion and deletion of nodes in the tree are implemented in such a way that their timing cannot be exploited for covert timing channels. In particular, tree operations should be completed within a clock tick. For if this were not the case, a high user's computation can maliciously cause the tree to grow to a considerable size by causing a lot of nodes to be inserted into it. A low-level computation generating fork requests may now experience observable delays due to the increased time taken by the session manager to update and manage the tree.

A possible solution to deal with the above scenario would be for the TCB to do the tree operations at random intervals. An approach that pursues a similar idea

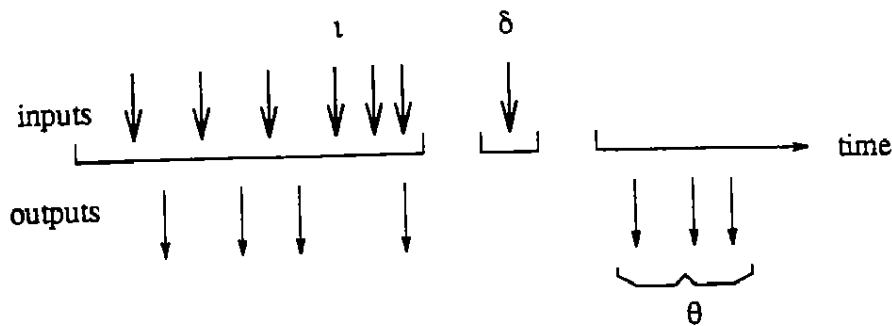


Figure 5.6: Illustrating the noninterference proof

to address hardware timing channels is based on the technique of *fuzzy time* [Hu91]. Fuzzy time techniques reduce the bandwidth of timing channels by adding noise to all sources of timing information and by ensuring that inputs and outputs are delivered at random intervals. We do not consider such solutions as they would take us beyond the scope of this thesis.

We now formally state and prove that the session manager is noninterfering.

Theorem 5.3 *In scheduling various concurrent computations, the session manager process is noninterfering.*

Proof:

The proof is by induction on the length of the input strings accepted by the session manager.

Basis: Consider the basis with input strings of length 1. It follows that by accepting only one input at a single security level, say l , the corresponding outputs will be at l or higher. This does not influence the outputs at the lower-view of l and it follows that the session manager will be noninterfering. Thus the basis holds trivially.

Inductive Step: For the induction hypothesis assume that for all input strings of length n , the session manager is noninterfering. For the inductive step, consider any given input string, say α , of length $n + 1$ which produces interference. Let δ be the

$(n + 1)^{\text{st}}$ input in this string (see figure 5.6). Also, let this interference be observed at level τ . By this we mean that there is at least one input at τ , and the $\text{purge}(\alpha, \tau)$ will cause the outputs in the lower-view of τ to change. Now let the outputs generated by the scheduler after the reception of the input δ , belong to the set θ with the individual outputs in the set denoted as $\theta_1, \theta_2, \dots, \theta_k$. From our earlier discussion on the two-step processing cycle of the session manager, it should be clear that the levels of the individual outputs in the set θ dominate the level of δ (we denote this as $l(\theta_j) \geq l(\delta), \forall j, j = 1, \dots, k$).

Now consider all inputs preceding δ . Note that an input at a level can only interfere with outputs in the lower-view of the level. So let us pick the most recent input, say ι at level τ , that could interfere with outputs in the set θ that are in the lower-view of τ . We must now look at the interaction of ι and δ . There are two cases of the input δ that we must now consider. The first one being the case where δ is a fork input event, and the second where δ is a term input event.

For the first case where δ is a fork, let us analyze the procedure **fork-trusted-agg** in figure 5.2. In this procedure, we determine if the forked computation generating δ can be immediately started or if it has to be queued for future execution. This is done by initiating a depth-first traversal of the session tree starting at the leftmost path and ending at the node representing the computation. Now in general, a computation f is denied immediate execution (startup) only if such a traversal encounters at least one nonterminated non-ancestor computation at or below the level of f before the node for f is traversed. Also, it follows from the requirements of serial correctness that the computation generating the input δ would have to be to the right of the one generating ι . (If this were not the case, the computation that would generate ι would

be suspended, and as such, there would be no input ι before δ .) Hence, during the depth-first traversal, the computation associated with ι would be encountered before the computation generating δ . Now if we purge the computation that generated the input ι , the outcome from the traversal of tree will be unaffected. In other words, if the computation associated with δ was denied execution, or allowed to start, this will continue to be the case after the purge. In other words, the output generated by the scheduler in response to input δ would remain the same for output events in θ at the lower-view of τ . Also, the purge will not generate any additional outputs in θ . We can thus conclude that these sequences do not lead to any interference.

Now consider the second case when δ is a term event. Once again, it follows from our invariants and the requirements of serial correctness that the computation generating the the term event δ would have to be the right of the computation generating the high input event ι , in any session tree. If this were not the case (i.e., if the low computation was to the left of the high computation), the high computation would not be executing due to serial correctness restrictions and thus cannot generate any high inputs. Now let us look at the procedure **terminate-trusted-agg** in figure 5.4 to see how terminate input requests are processed. We observe that when a computation say t , terminates, a depth-first traversal of the session tree is initiated to identify potential leaf computations to the right of t , that could be released for execution. A leaf computation in the tree is started only if there exists no previously traversed active or queued computation at or below the level of the leaf computation. Now in a depth-first search, the computation generating ι will be encountered before the one generating δ . However, the computation generating ι is at a higher or incomparable level with respect to the computation generating δ , and thus the purge of the former will not affect the outcome of the traversal. Thus there is no interference as the out-

put events in θ that are generated in response to δ would remain the same. To put it more precisely, the output events in the lower-view of τ that are in θ , remain the same.

Thus far, we have shown that the input ι does not interfere with outputs in the set θ and in the lower-view of τ . Let us now proceed by purging the input ι from the original string of length $n + 1$ to get string n' . We will now get a new output set θ' . The set θ' may differ from θ only in that it doesn't contain the outputs of ι which are at the level of ι or higher. Thus, events in the lower-view of τ would remain unchanged in both sets θ and θ' . Now since we just demonstrated that the input ι causes no interference, it follows that if the original string of length $n + 1$ is interfering, this interference must be also observable in the outputs of the string n' with ι purged. In other words, this interference must be observable in θ' which retains all outputs below τ in the original output set θ . However, the string n' is of length n , and if it is interfering, will contradict the induction hypothesis which assumed the session manager is noninterfering for *all* strings of length n . Hence the proof for the induction step. \square

Chapter 6

Kernelized Architecture Implementation

In this chapter we discuss the implementation of our scheduling algorithms within the framework of a kernelized architecture [TS92]. We begin by discussing the architecture. This is followed by a detailed description of the conservative and aggressive scheduling algorithms. We finally end the chapter with the major theorems and proofs. Unlike the trusted subject architecture, the kernelized one requires no proof for confidentiality.

6.1 Architecture

As mentioned before, kernelized architectures in general have no trusted subjects. In our architecture, there exists a single-level level manager process at each level to coordinate the various concurrent computations running at the respective levels. Figure 6.1 illustrates the kernelized architecture.

While security comes for free in a kernelized architecture (since there are no trusted subjects), the price we have to pay is the additional complexity involved in implementing the various scheduling algorithms. In particular, the lack of a global multilevel coordinator such as the session manager in the kernelized architecture, necessitates that we implement the scheduling schemes in a distributed fashion.

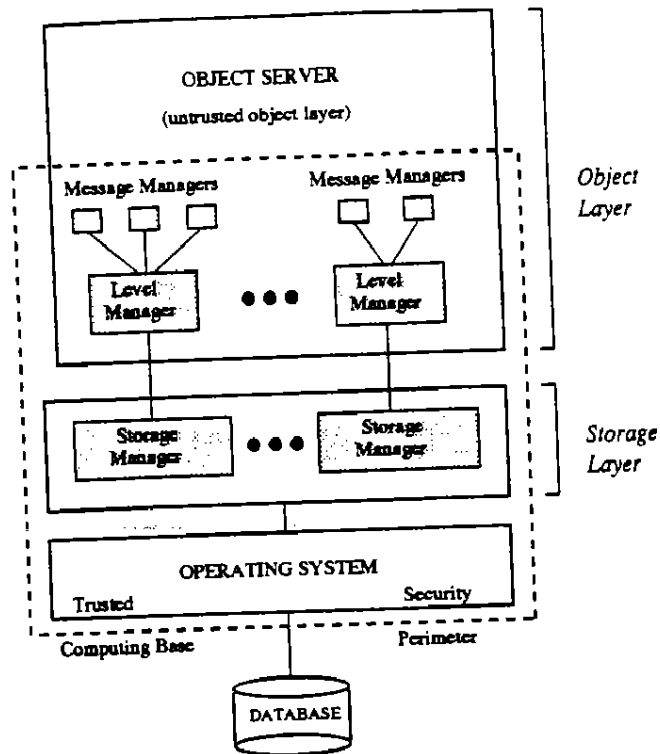


Figure 6.1: A kernelized architecture

6.2 Scheduling Algorithms

We now describe the algorithms to implement both the conservative and aggressive scheduling schemes under a kernelized architecture. We begin by describing some level manager data structures that are common to both schemes.

Level manager data structures:

- current-wstamp:** the current timestamp given to objects written at the level of the level manager;
- queue:** a queue of message managers waiting to be activated;
- terminate-history:** a list of ordered pairs (forkstamp, wstamp);

In addition to the above data structures for every level, a message manager utilizes the following:

```

Procedure terminate-kern-cons(lmsgmgr, wstamp, forkstamp)
{
  %Let tt be the message manager that just terminated at level lmsgmgr
  %Let lm be the level manager at level lmsgmgr

  %Update local current write stamp from tt
  lm.current-wstamp ← wstamp

  %Update local Terminate-history with the forkstamp and wstamp of tt
  Append-terminate-history(terminate-history, forkstamp, wstamp);

  If queue is not empty
  then
    dequeue(queue, mm);
    start(mm);
  Else
    Send a WAKE-UP message to all immediate higher level managers;
  End-If
}
end procedure terminate-kern-cons;

```

Figure 6.2: Level manager algorithm for terminate processing

local-stamp: a local table of timestamp entries with an entry at each level dominated by the message manager and identifying the versions at the level that will be used to process read-down requests;

forkstamp: forkstamp issued by the parent message manager;

6.2.1 Conservative Scheduling Algorithms

We now discuss the algorithms to implement the conservative scheduling scheme. Let us begin by looking at how fork requests are processed. When a computation is forked, a new message manager is created, and it is unconditionally queued by the local level manager, as shown in procedure `fork-kern-cons` in figure 6.3. The procedure also initializes the forkstamp entry and the phase 1 entries of the local-stamp table passed on by the ancestors (this is the `astamps` parameter for the procedure). When a level manager is notified of the termination of a message manager at its level, it first

```

Procedure fork-kern-cons(level-parent, level-create, forkstamp, astamps)
{
  %Let level-create be the level of the local message manager
  Create a new message manager mm at level level-create;

  %Record the forkstamp passed on by the parent
  mm.forkstamp ← forkstamp

  %Begin phase 1 of acquiring local-stamp entries
  For (every level l ≤ level-parent)
  do
    initialize mm.local-stamp table entries from astamps;
  End-For

  %This is a priority queue maintained in forkstamp order
  enqueue(queue, mm);
}
end procedure fork-kern-cons;

```

Figure 6.3: Level manager algorithm for fork processing

```

Procedure wake-up-kern-cons
{
  %Proceed if the necessary condition has been met
  If a WAKE-UP message has been received from all lower levels
  then
    If the queue is not empty
    then
      dequeue(queue, mm);
      start-kern(mm);
    else
      Send a WAKE-UP message to all immediate higher levels;
    End-If
  End-If
}
end procedure wake-up-kern-cons;

```

Figure 6.4: Level manager algorithm for wake-up processing


```
Procedure start-kern(nn)
{
  %Let nn represent the message manager to be started
  %Let lm represent the level manager managing nn

  %Complete phase 2 of acquiring local-stamp entries
  For (every level l lower than the level of nn for which no timestamp
  has been obtained so far)
  do
    nn.local-stamp[l] ← mm.wstamp;
    where mm is the message manager entry in the terminate-history at level
    l
    with max{forkstamp: forkstamp < nn.forkstamp}
  End-For

  %Update the write stamp (wstamp) from the level manager
  nn.wstamp ← lm.current-wstamp + 1;

  %Begin execution of the message manager nn
  execute(nn);
}
end procedure start-kern;
```

Figure 6.5: Level manager algorithm for **start** processing

updates the local **terminate-history**. The level manager then dequeues and starts the next computation at the head of its local queue; if the queue is found to be empty, a WAKE-UP message is sent to all immediate higher levels (see procedure **terminate-kern-cons** in figure 6.2). Let us now see how WAKE-UP messages are processed. When a level manager receives a WAKE-UP message from each of the immediate lower levels in the security lattice, it dequeues its local queue and starts the next message manager; if the queue is empty, the WAKE-UP message is simply forwarded to all the immediate higher levels in the lattice, as shown in the procedure **terminate-kern-cons** in figure 6.2.

As in the trusted subject implementation, a message manager's **local-stamp** vector is once again initialized in two phases. The first phase entries are obtained as before from the ancestors. The second phase utilizes the **terminate-history** data structures at all levels dominated by the level of the starting message manager. Recall that this history contains a list of terminated computations identified by their forkstamps and their associated **wstamp** values at termination time. At each level, the computation with the largest forkstamp that is still less than the forkstamp of the message manager to be started, is selected, and the associated timestamp is read into the corresponding **local-stamp** entry.

We conclude this subsection by giving proofs of correctness and termination for our conservative level-by-level scheduling algorithms.

Proof of correctness.

Theorem 6.1 *The conservative (level-by-level) scheduling algorithms for the kernelized architecture maintain the invariant **inv-conservative**.*

Proof:

While there are two message manager algorithms, namely **send** and **quit** and four level manager algorithms **fork**, **start**, **terminate** and **wake-up**, we focus only the latter two for the proof. The **terminate** and the **wake-up** algorithms invoke the **start** procedure whereby computations get activated (started). It suffices therefore to show that these algorithms maintain the invariant **inv-conservative**.

Consider the **terminate** procedure first. If we assume that the invariant holds as a pre-condition before the procedure was invoked, then it follows that there are no active or queued computations at level **lmsgmgr** or lower. Now if the **start-kern(mm)** statement is reached, the following pre-conditions are true: (a) there exists one or more queued computations at level **lmsgmgr**; (b) the computation **mm**, with the lowest **forkstamp** will be started. The **start-kern(mm)** statement further ensures the post-condition: (c) **mm**, being the computation with the smallest **forkstamp** is started, and there are no queued or active computations at lower levels. This maintains the invariant. On the other hand, if the **start-kern(mm)** statement is not reached the invariant obviously continues to be true.

Consider the **wake-up** procedure next. From the **terminate** procedure we see that a **WAKE-UP** message is sent to all immediate higher levels only if there are no active or queued computations at or below the level that sent the message. Hence, when a **WAKE-UP** message has been received at a level say **lwake**, from all lower levels, the following are true:

- d. there are no queued or active computations at levels lower than **lwake**;
- e. there are no active or terminated computations at level **lwake**.

The latter condition is true since a computation can be started only as a result of a previous **terminate** at the same level or due to the receipt of a **WAKE-UP** message

(and no terminate event would have occurred at `lwake` at this point). Thus when the statement `start-kern(mm)` is executed, the following post-condition is true: (f) `mm` is the first computation to be activated at level `lwake` and there exists no queued or active computations at levels `lwake` or lower. This clearly maintains the desired invariant. Once again, if the `start-kern(mm)` is not reached, the invariant continues to hold. \square

Having shown how our algorithms maintain the invariant **inv-conservative**, we now argue how these algorithms preserve serial correctness by maintaining correctness constraints 1, 2, and 3. We state this below as a corollary.

Corollary 6.1 *The conservative (level-by-level) scheduling implementation under invariant **inv-conservative** maintains serial correctness.*

Proof:

When a computation `c` is started at a level `l`, the invariant **inv-conservative** requires all computations that are forked at level `l` with smaller forkstamps, to have terminated. This maintains correctness constraint 1. The invariant also requires that on the start of computation `c`, all computations at levels lower than `l` to have terminated. This requirement clearly maintains correctness constraint 2 since the constraint requires only computations with smaller forkstamps than `c` and at levels `l` or lower to have terminated. In other words, as far as lower level computations are concerned, the invariant **inv-conservative** is more restrictive than correctness constraint 2, and clearly maintains and implies the latter. Correctness constraint 3 has to do with versions assigned to process read-down requests. The arguments given for the accumulation of **local-stamp** entries in the trusted subject architecture still apply. Thus the **local-stamp** table entries collected in the first phase at fork time by `c` reflect versions identifying the states of objects written at the level of these

ancestors before each successive child in the ancestral path was forked. The second phase entries as before, identify latest versions written at lower levels for which there were no ancestors. In summary, all read down operations that are mapped to the versions identified by the **local-stamp** entries, will read the same object states as in a sequential execution, and thus maintain correctness constraint 3. \square

Proof of termination

We formally state as a theorem, that a session will eventually terminate.

Theorem 6.2 *Under the conservative scheduling scheme, all computations in a session will eventually terminate and thus guarantee the termination of a user session.*

Proof:

By induction on the number of security levels, n , at which computations are forked in a session.

Basis: Consider the basis with $n = 0$. Then the only level with active computations will not have any fork requests emanating from it. It follows from the second part of the proof of lemma 5.1 that the session is guaranteed to terminate.

Inductive Step: For the induction hypothesis assume that when n is equal to m , all computations terminate at the m levels and a WAKE-UP is sent to all immediate higher levels. For the inductive step consider $m + 1$ levels where level l_{m+1} is a maximal element in the security lattice and dominates a subset of the m levels. Now by the induction hypothesis, all computations at the m levels would have terminated and hence a WAKE-UP message would have been received at level l_{m+1} from all immediate lower levels in m . It now remains for us to show that a WAKE-UP is received at level l_{m+1} from all immediate lower levels dominated by l_{m+1} that never had active computations in the user session. These levels thus do not belong to

m . The argument to show this can be made from the following: (1) The induction hypothesis guarantees that the root computations which are at the lowest level, say l_1 , in m , would have terminated and sent a WAKE-UP message to all immediate higher levels; (2) WAKE-UP messages are always forwarded across empty levels. Hence all levels which dominate l_1 and in turn are dominated by l_{m+1} would have WAKE-UP messages forwarded through them. This guarantees that l_{m+1} would receive these messages from all immediate lower levels, and when this happens the computation at the head of the queue which has the smallest forkstamp will be dequeued and started. The termination of this first computation at level l_{m+1} is again guaranteed by lemma 5.1 and leads to the startup of the next one in the queue. Every terminate results in the next computation in the queue to be subsequently started in turn. The queue will thus be progressively emptied in finite steps and all computations at level l_{m+1} would have then terminated. Thus the entire session will terminate. \square

6.2.2 Aggressive Scheduling Algorithms

Having discussed a conservative scheduling scheme, we now turn our attention to an aggressive scheduling scheme. The implementation algorithms for the aggressive scheduling scheme are given in figures 6.6, 6.7, and 6.8 (the start algorithm is the same as in figure 6.5 for the conservative scheme). In addition to the data structures needed to implement the conservative scheme, the aggressive one requires that every level manager maintain a fork-history consisting of a list of ordered pairs (forkstamp, level). This helps a level manager keep track of the fork requests generated at its level. We now elaborate on these algorithms.

When a computation is forked (see the if statement in figure 6.6), we have to decide if it can be started immediately. A forked computation is started immediately if there exists no non-terminated computations at lower levels and with smaller

```

Procedure fork-kern-agg(level-parent, level-create, forkstamp, astamps)
{
  %Let level-create be the level of the local level manager
  Create a new message manager mm at level-create;

  %Record the forkstamp passed on by the parent
  mm.forkstamp ← forkstamp

  %Begin phase 1 of acquiring local-stamp entries
  For (every level  $l \leq$  level of the parent of mm)
  do
    initialize mm.local-stamp table entries from astamps;
  End-For

  %Check to see if a forked computation can be started immediately
  If  $\forall l \leq \text{level}(mm), \neg \exists$  any computation  $c : (c.\text{forkstamp} < mm.\text{forkstamp}$ 
     $\wedge c \notin \text{terminate history at } l)$ 
    then,
    start-kern(mm);
  else
    %This is a priority queue maintained in forkstamp order
    enqueue(mm);
  end-if
}
end procedure fork-kern-agg;

```

Figure 6.6: Processing fork requests under aggressive scheduling

```

Procedure wake-up-kern-agg
{
  dequeue(queue, mm);
  start-kern(mm);
}
end procedure wake-up-kern-agg;

```

Figure 6.7: Processing wake-up requests under aggressive scheduling

```

Procedure term-kern-agg(lmsgmgr, wstamp, forkstamp)
{
  %Let tt be the message manager that just terminated at level lmsgmgr
  %Let lm be the level manager at level lmsgmgr

  %Update local current write stamp from tt
  lm.current-wstamp ← current-stamp

  %Update terminate history
  Append-terminate-history(terminate-history, forkstamp, wstamp)

  %Check if a computation at level lmsgmgr can be started
  If queue is not empty
  then
    %Let mm be the computation at the head of the queue
    If  $\forall l < \text{level}(mm), \neg \exists c : (c.\text{forkstamp} < mm.\text{forkstamp} \wedge c \notin \text{terminate history at level } l)$ 
    then
      dequeue(queue, mm);
      start(mm);
    End-If
  End-If

  %Check if a computation at levels  $\geq lmsgmgr$  can be started
  For all levels  $l \leq lmsgmgr$ 
  do
    If  $\exists c \in \text{fork-history at } l \text{ with } (\text{level}(c) > lmsgmgr \wedge c.\text{forkstamp} > tt.\text{forkstamp}) :$ 
     $\neg \exists$  any computation k with  $(\text{level}(k) \leq \text{level}(c) \wedge k.\text{forkstamp} < c.\text{forkstamp} \wedge k \notin \text{terminate history at level}(k) \wedge k \text{ is not an ancestor of } c)$ 
    % We checked to see if c was not preceded by a lower-level active or queued
    % non-parent computation in any of the fork-histories searched
    then
      Send a WAKE-UP message to the level manager of c at level l;
    End-If
  End-For
}
end procedure term-kern-agg;

```

Figure 6.8: Processing terminate requests under aggressive scheduling

forkstamps. We can determine all the computations forked at lower levels by examining the fork histories at these levels. We can further determine which of these computations have terminated by examining the terminate histories at these lower levels. When processing terminate requests, a similar check is made upon the termination of a computation at a level to see if the next computation, if any, at the head of the queue for this level, can be started (see figure 6.8). We also check to see if computations queued at higher levels can be released. We examine the fork histories at lower levels for computations that have been forked from these lower levels but have larger forkstamps than the just terminated computation (see the for statement in figure 6.8). Such computations with larger forkstamps can be started so long as they are not preceded by lower level non-terminated computations to the left, in the computation tree. A WAKE-UP message is sent to the level managers at the levels for which computations can be started. On receiving such a message, a level manager dequeues and starts the next computation at the head of its queue (see figure 6.7).

We now give proofs of correctness and termination for the aggressive scheme.

Proof of correctness.

Theorem 6.3 *The aggressive scheduling algorithms for the kernelized architecture maintain the invariant inv-aggressive.*

Proof:

We start with the fork-kern-agg procedure in figure 6.6. We see that for the statement start-kern(mm) to be executed, the following pre-conditions are true:

- a. there exists no non-ancestral queued or active computations at or below level(mm) and with a smaller forkstamp than mm;
- b. mm is the only computation at level(mm).

After computation mm has been started the condition (a) above still holds and thus the invariant is maintained. A similar argument can be made for the `start-kern(mm)` statement in procedure `term-kern-agg`. When mm is dequeued, condition (a) above holds, and since mm has the smallest forkstamp in the queue, the invariant is maintained after the execution of `start-kern(mm)`.

It now remains to show that the start-up of a computation due to the receipt of a WAKE-UP message at a level, will not violate the invariant. To see this, we observe that a WAKE-UP message is sent to a higher level (in the `terminate-aggressive` procedure) only if there exists a pending computation say, c at the higher level that was denied immediate execution at fork time. Further, c has to have the smallest forkstamp among others at its level and should not be preceded by active or queued computations at lower levels and with a smaller fork than itself. Thus on receiving a WAKE-UP message, a level meets all the necessary conditions to start a computation. The post-condition following the `start-kern(mm)` statement in procedure `wake-up-aggressive` thus maintains the invariant. \square

We now state and show how the invariant `inv-aggressive` maintains serial correctness under our implementation.

Corollary 6.2 *The aggressive scheduling implementation maintains serial correctness.*

Proof:

We basically have to show how the correctness constraints 1, 2, and 3 are maintained. For a computation to be dequeued and successfully started, invariant `inv-aggressive` requires all earlier forked computations at level l or lower, to have terminated. But this is what is precisely required to maintain correctness constraints 1 and 2. The

argument for the maintenance correctness constraint 3 is independent of the scheduling algorithm used. Thus the earlier argument given for the conservative case still holds. \square

Proof of termination

Theorem 6.4 *Under the aggressive scheduling scheme, all computations in a session will eventually terminate and thus guarantee the termination of a user session.*

Proof:

To argue proof of termination for the aggressive algorithm, we observe that if a computation is denied immediate execution this can only be at fork time. Again we assume that once started, a computation is guaranteed to terminate (by lemma 5.1). Our task is thus basically to show that every queued computation will eventually be started. Now if on fork, a computation f is denied immediate execution, then there must be at least one active computation say c , with a smaller stamp than f and at or below level(f). Now the termination of c is guaranteed by lemma 5.1. The termination of c will cause at least one computation with a greater forkstamp than c and a smaller forkstamp than f , or f itself, to be started. Now if f is not started, there can only be a finite number of computations that can potentially block f . Subsequent terminate events will progressively decrease the number of such computations with a smaller forkstamp than f . This will result in the eventual release of f for execution. With a similar argument, we can show that every queued computation will eventually be released for execution and thus run to termination. Thus the entire session will eventually terminate, concluding the proof. \square

Chapter 7

Replicated Architecture Implementation

In this chapter we discuss the implementation of the message filter model and related scheduling schemes within the context of the replicated architecture [TS94b]. We begin by reviewing the architecture itself. This is followed by a discussion of message filtering issues from an architectural standpoint, as well as a some elaboration on the interplay between serial correctness, scheduling, and replica control. We then present the various scheduling algorithms, and the related theorems.

7.1 Architecture

The replicated architecture for multilevel secure database management systems (mls DBMS's) has lately experienced a resurgence in the research community. As mentioned before, it represents one of the three architectures identified by the Woods Hole study organized by the U.S. Air Force [Cou83]. The distinguishing feature of the replicated architecture is that lower level data is replicated at higher levels. To be more precise, for any given security level, a physically separate DBMS is used to manage data at or below the level. In our further discussions, we use the term "container" to be synonymous with "database". These backend databases are untrusted, and rely on a trusted front end (TFE) that hosts the trusted computing base (TCB), for access mediation.

The replicated architecture, as elaborated for a simple lattice, is shown in

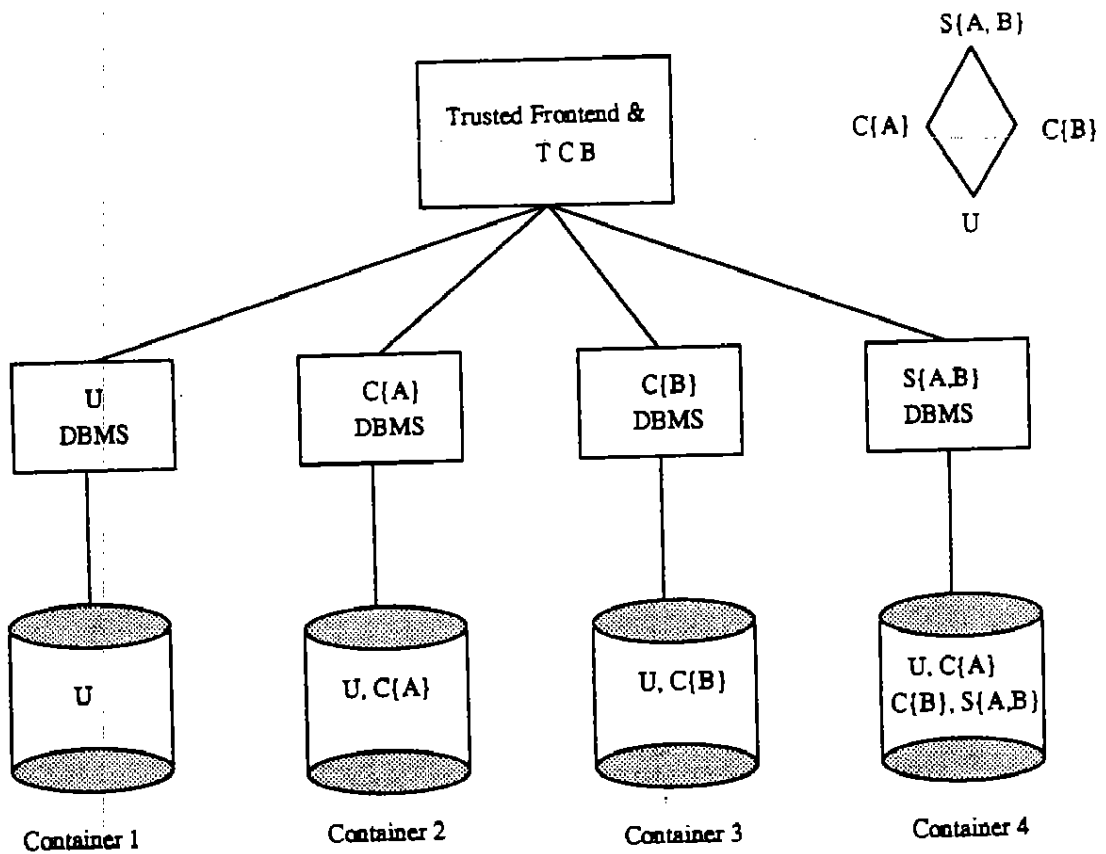


Figure 7.1: The replicated architecture illustrating containers for a simple lattice

figure 7.1. The objects classified at U and stored in the first container are replicated across the other containers 2, 3, and 4. Such replicas when stored at containers 2, 3, and 4, are no longer considered to be at level U, rather are classified at the level of their respective containers. The advantages and security of the replicated architecture stem from the fact that users (or subjects acting on their behalf) at different levels are physically isolated from one another, and that a user is able to accomplish all tasks (multilevel queries and updates at his or her level) from the data stored at a single DBMS. This is because a properly cleared user who logs in to the system at security level l , will be assigned to the DBMS at l . All data that is classified below levels l and stored at the lower level databases is replicated, and thus available, at the DBMS at l . Thus, for example, security threats from covert channels due to read-down operations in multilevel queries do not arise in this architecture.

The benefits of the replicated architecture come at the cost and complexity of the replica control schemes needed to keep the replicas of the data mutually consistent. To make this architecture commercially viable, these schemes would not only have to be efficient, but in addition must be secure (in that they do not introduce covert channels). It is important to note that covert channels can be introduced in this architecture only through the TFE. Replica control algorithms for relational databases under the replicated architecture have appeared in recent literature [Cos92, CM92, KJ90, MJS91]. The work reported here is the first to address these issues within the context of object-oriented databases. With the ever increasing interest in object-oriented databases, our effort here is timely, and we hope, will provide impetus for further work in the area.

7.2 Message-filtering in the Replicated Architecture

When we consider the implementation of message filtering in the replicated architecture, the very nature of the architecture poses a different and unique set of problems. We have to deal with security and integrity aspects of processing data within a single container as well as multiple containers.

Consider first the issues pertinent to a single container. The way objects are replicated and classified at the various containers, and the fact that only a subject cleared to the level of a container can access the data at the container, have the following implications:

1. There exists no need for message filtering between objects at a single container.
2. Method invocations resulting from messages sent between objects at a single container can be processed sequentially, as there exists no downward signaling channel threat.
3. There exists the need for integrity mechanisms to prevent replicas at a single container from being updated arbitrarily by subjects.

In other words, we do not enforce any message filtering or mandatory security controls between objects at a single container, since doing so would require access mediation mechanisms to be imported into the individual backend DBMS's. This clearly goes against the original spirit and motivation of the replicated architecture. Messages sent from low replicas to other objects within a single container result in method invocations which are processed sequentially according to RPC semantics. Hence there is no need to maintain multiple versions of objects. Also, covert channel threats do not exist, as only subjects cleared to the level of a container can observe the results of local computations. Finally, the lack of mandatory controls within

a container has to be balanced with adequate integrity mechanisms giving us the assurance that such replicas will not be updated by the local subjects at a container.

In contrast to the above, dealing with objects residing at different containers does require message filtering so as to prevent illegal information flows. If we review the different filtering cases in the message filtering algorithm (as shown in figure 3.1), we now see that case (4) which deals with messages sent from higher level to lower level objects, is degenerate. This is because messages sent downwards in the security lattice to enable read-down operations do not cross the boundary of a container, and as mentioned before, involve no filtering. Messages sent to higher and incomparable levels will still need to be filtered. In particular, when messages are sent to higher objects (residing at higher level containers), concurrency may again arise.

Having discussed the message filtering and security issues in the replicated architecture, we now turn our attention to the trusted computing base (TCB) in the architecture. As mentioned before, the TCB is hosted within the trusted front end (TFE). A design objective in any secure architecture is to minimize the number of trusted functions that need to be implemented within the TCB. This enables the TCB to have a small size, and thereby making its verification and validation easier. In light of this, is it possible to implement the various coordination and replica control algorithms while keeping the size of the TCB small? In later sections of the chapter, we present replica control and coordination schemes that require minimal functionality from the TCB. To be more precise, the role of the TCB reduces basically to that of a router of messages from lower level to higher containers. In particular, the TCB requires no trusted (multilevel) subjects or data structures. All scheduling and coordination is achieved through single-level subjects at the backend databases. In other words, this portion of the front-end TCB could be implemented using a kernelized architecture.

7.3 Serial Correctness and Replica Control

Recall from our previous discussion that sending messages between objects at a single container involves no message filtering, while sending messages to objects across containers does call for filtering. When filtering is involved concurrency is once again inevitable and we have to ensure that the concurrent computations executing across the various containers preserve serial correctness. We now investigate the interplay between serial correctness, the various scheduling algorithms, and replica control.

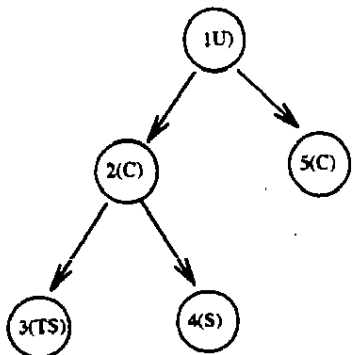
In chapter 4, three constraints were presented as sufficient conditions to guarantee serial correctness of concurrent computations. Correctness constraints 1 and 2 are required to govern the scheduling of concurrent computations while the third constraint governs how versions should be assigned to process read-down requests. Constraints 1 and 2 would now have to be interpreted for computations executing across containers. For example, when a computation c is started at a level l (container C_l), constraint 2 would now read: *All current non-ancestral as well as future executions of computations that have forkamps smaller than that of c , would have to be at containers for level l or higher.* Also, the fact that there are no trusted subjects in our implementation means that there will be no central coordination of the computations executing across the various containers. Hence the implementation of the various scheduling algorithms would have to be inherently distributed, as in the kernelized architecture presented earlier. Finally, correctness-constraint 3 also has to be reinterpreted for the replicated architecture as we no longer maintain versions of objects. The original requirement that a computation c reading down obtain the versions of lower level objects consistent with a sequential execution, now maps to the requirement that the various updates (also called update projections in the literature) producing these different versions be shipped and applied to c 's container before it starts executing. This last constraint thus has a direct implication on the

replica control schemes that would be utilized for the architecture.

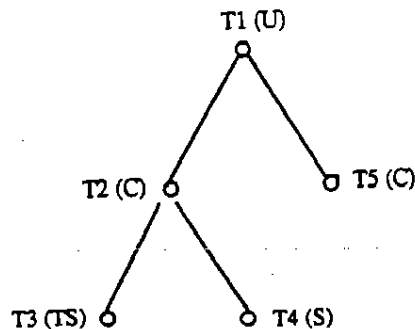
In order to reason about update projections and their effect on serial correctness, we introduce the notion of *r-transactions*. This is done only for ease of exposition. Our solutions do not impose or mandate any particular model of transactions. Transactions allow us to conveniently group sequences of updates, and in particular those that need to be incrementally propagated to higher containers. We use the prefix "r" to distinguish this notion of transactions with other models that will be used in the rest of the thesis. We drop the prefix when it is clear from the context that we are referring to r-transactions.

In the object model of computing, every message in is received at an object and results in the invocation of a method defined in that object. We refer to such an object as the home object of the method. The subsequent activity (reads and updates) within the boundary of a home object can be modeled as belonging to a r-transaction. Every message in a message chain can be mapped to a corresponding transaction. This leads to a hierarchical (tree) model of transactions for a user session. We consider the root message as starting a root transaction. The root transaction in turn issues other transactions which we see as its descendants in the tree. Figure 7.2 illustrates the transaction tree for a computation tree.

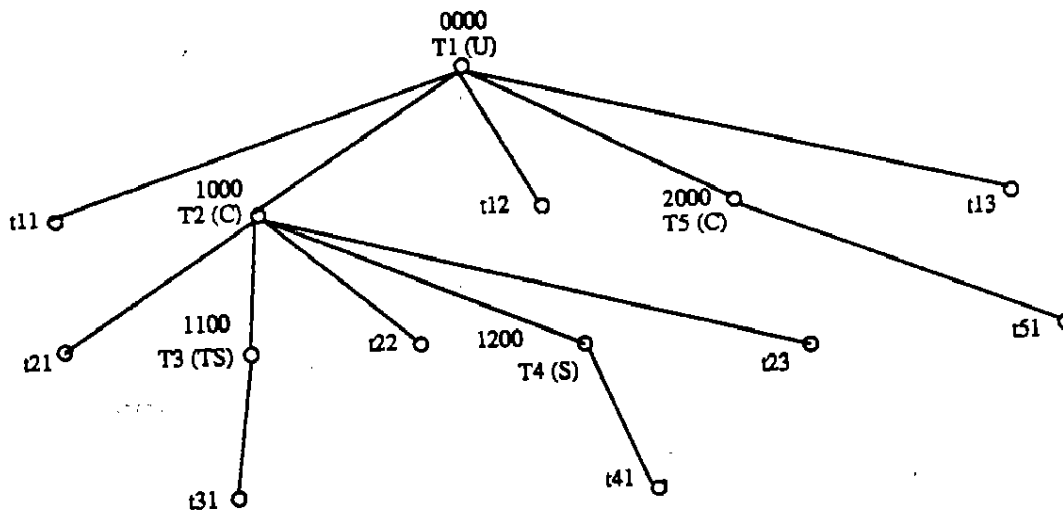
A depth-first (left-to-right) traversal of a transaction tree starting with the root transaction, will give the sequence in which the transactions are issued and started within a user session. To illustrate how serial correctness is to be maintained within a session and in the context of the replicated architecture, we need to zoom in and take a magnified look at the transaction tree. This is because a transaction may make its partial results visible to other transactions at different containers. Consider any subtree in figure 7.2 such as the one rooted at transaction T_2 . A child of T_2 , such as T_3 , is allowed to see (read down) only part of the updates made by T_2 . To



A computation tree



A transaction tree



The corresponding subtransaction tree

Figure 7.2: A transaction tree and its subtransaction mapping

be more precise, it is only those updates made by T_2 up to the point T_3 was issued. The second child T_4 will be allowed to see all the updates seen by T_3 , and in addition those made by T_2 between the interval that T_3 and T_4 were issued.

To model and visualize partial visibilities within transaction boundaries, we introduce *r-subtransactions* as finer units of transactions. The second and larger tree in figure 7.2 illustrates a subtransaction tree derived from the original transaction tree. A transaction such as T_2 is now chopped up into three subtransactions $t_{2,1}$, $t_{2,2}$, $t_{2,3}$. The subtransaction $t_{2,1}$ represents all the updates by T_2 until transaction T_3 was issued. Subtransaction $t_{2,2}$ similarly represents the updates between the interval that transactions T_3 and T_4 were issued. Finally, the subtransaction $t_{2,3}$ accounts for all the remaining activity in T_2 before it committed. A subtransaction is seen as having a relatively short lifetime, and is required to commit before any sibling subtransactions to the right, or child transactions (and implicitly subtransactions) are started. The operations issued by a subtransaction are said to be *atomic operations*. Such operations never cross the boundary of their relevant home object and cannot lead to the sending of further messages (or the issuing of transactions) to other objects. Serial correctness requires that an individual transaction, such as T_4 see all the updates of all subtransactions below its level that will be encountered in a depth-first search of the subtransaction tree starting with the root and ending in T_4 . Thus for T_4 this will include subtransactions $t_{1,1}$, $t_{2,1}$, and $t_{2,2}$. The updates of all these subtransactions except $t_{2,2}$ would have to be seen by the left sibling of T_4 , which is transaction T_3 , and thus would have already been applied logically at the relevant containers before T_4 was issued.

We formally define these and other notions below:

Definition 7.1 We consider a subtransaction to be a totally ordered set of atomic operations. We define a transaction T_i to be a partial order $(s_i, <_{T_i})$ such that:

1. s_i is a set of operations, and each operation may be a subtransaction or another transaction $T_j = (s_j, <_{T_j})$.
2. The relation $<_{T_i}$ orders at least all conflicting atomic operations in s_i .

Definition 7.2 We define the replica-set of a transaction T_j at level j to be the set of updates of subtransactions at or below level j that will be seen by T_j in a sequential execution (or depth-first search) of the tree.

Definition 7.3 We define the propagation-list of a transaction T_j to be those updates in the replica-set of T_j that have not been seen by T_i , where T_i was the last transaction that was issued before T_j in a sequential execution. These updates are those made by subtransactions at levels lower than j , and to the right of T_i and to the left of T_j (in the subtransaction tree).

In figure 7.2, the replica-set of transaction T_4 will consist of the updates issued by subtransactions $t_{1,1}$, $t_{2,1}$, and $t_{2,2}$. The propagation-list of T_4 will consist of the updates issued by subtransaction $t_{2,2}$.

Now in the replicated architecture, the transactions in a subtransaction tree execute across containers. Whenever an object in a low container issues a write-up request, a message will be sent upwards in the lattice, and routed by the TFE to the appropriate high level container. Such a message will be received by an object in the higher level container and eventually result in the invocation of a method. Before this method can be invoked (i.e., before the corresponding transaction can be started), we need to do the following:

1. Determine if it is safe to begin execution of the transaction;

2. Make sure that the propagation-list of the transaction has been applied at the local container.

The first consideration above arises from the fact that the transactions (methods) generated by a session execute across the various containers in a distributed fashion, and this may lead to transactions at higher containers starting prematurely (when compared to centralized sequential execution). We thus require the start-up of transactions to be governed by some invariant. Once a transaction is allowed to start (i.e., doing so would not violate the invariant), the replica control scheme should ensure that the relevant propagation-list (set of update projections) is applied at the local container of the transaction.

The global serial order implied by the tree is not known to the containers, and has to be derived by labeling transactions with forkstamps as shown in figure 7.2. The hierarchical scheme proposed earlier in chapter 4 can be used to derive such forkstamps. Thus in figure 7.2, the root transaction is assigned an initial forkstamp of 0000. Every descendant transaction that is assigned a stamp that is derived from this initial one by progressively incrementing the most significant (leftmost) digit by one. To generalize this scheme for an entire transaction tree with many transactions, we require that with increasing depth in the tree, a less significant digit be incremented. When a new transaction is created, its propagation-list is also assigned the same forkstamp as the transaction. Also, the last subtransaction issued by a transaction is given a forkstamp that would have been given to the next child transaction had there been one.

Before concluding our discussion on serial correctness and replica control, we note that in the replicated architecture serial correctness alone is insufficient to guarantee the mutual consistency of replicated data. This is because serial correctness can be guaranteed by shipping update projections only to the containers which have

forked transactions for a session. In other words, if a transaction was not forked for a level, the replicas at the container for the level could be out-of-date, and we would still not violate serial correctness. The scheduling algorithms that we present in the next section not only guarantee serial correctness, but in addition ensure that when a session terminates, all containers will be mutually consistent. When such consistency is guaranteed, we say that the algorithms preserve the *final-state equivalence* of all the containers.

7.4 Scheduling Algorithms

In this section we discuss how we can combine replica control, with the conservative and aggressive scheduling strategies. As in the kernelized architecture, the conservative scheme involves less complexity and is thus easier to implement. We begin by clarifying some aspects related to the execution and failure semantics of transactions, as well as some of the necessary data structures to be maintained by individual containers.

A *r*-transaction as described in this chapter, is characterized by the property of failure atomicity. Hence if any of the subtransactions of a transactions fails or aborts, we have to abort the entire corresponding transaction. This would also require that we undo the effects of any committed earlier subtransactions. To avoid this, and still guarantee failure atomicity, we allow a transaction to commit only if all its subtransactions commit. The updates of committed subtransactions are made permanent in the database only when the parent transaction commits. Also, we take the commit of the root transaction to imply that the entire session has committed.

To implement our scheduling strategies and replica control schemes, every container C_j at level j maintains the following data structures for an active user session:

- Activation-queue;: this is a priority queue of transactions that is maintained according to the forkstamps;
- Projection-queue;: a queue which stores update projections (propagation-lists) by their forkstamps;

When transactions start issuing other transactions at higher levels, the relevant propagation-lists (update projections) are incrementally shipped to higher containers and stored in their projection queues. When a scheduling scheme calls for a transaction to be started, it is dequeued from the local activation queue and the relevant update projections are applied to the container just before the transaction starts.

7.4.1 Conservative Scheduling Algorithms

Recall that the conservative scheduling scheme calls for computations to be executed on a level-by-level basis. Cast in terms of transactions, the conservative scheduling scheme maintains the following invariant:

Inv-conservative-replicated: *A transaction T is executing at a container C_l at level l only if all transactions at lower levels, and all transactions with smaller forkstamps at level l , have terminated.*

This invariant is basically the same as **invariant-conservative** presented in chapter 4, but is cast in terms of transactions. When a new transaction is forked, the container receiving the fork request invokes the procedure **fork-rep-cons** shown in figure 7.3. On being forked, a transaction is unconditionally queued in the activation queue of the local container. Also queued is the update projection (propagation-list) obtained from the parent transaction that issued the fork. The update projection is given the same forkstamp as the transaction just queued. The parent transaction also sends these update projections to all other higher containers.

When a transaction terminates, the processing steps involved are shown in

procedure **term-rep-cons** of figure 7.4. Upon termination, the updates of the last subtransaction are applied to the local container, and this is followed by the posting of these updates to all higher level containers. A higher level container on receiving these updates will queue them in its projection queue. The activation queue is then checked to see if there are any more transactions at the container that need to be started. If the queue is not empty, the next transaction at the head of the queue is started. If the queue is empty, the container knows that all transactions at lower containers have terminated, and no more update projections will be forthcoming. The container then empties any remaining entries in its projection queue and applies the updates, and sends a WAKE-UP message to all immediate higher level containers.

The processing of WAKE-UP messages is illustrated in procedure **wake-up-rep-cons** of figure 7.5. When a container receives a WAKE-UP message from all immediate lower level containers, it examines its activation queue, and if it is not empty, dequeues the transactions one by one in forkstamp order. On the other hand, if the queue is empty, the local projection queue is processed and a WAKE-UP message is forwarded to all immediate higher levels.

The processing of terminate and wake-up requests involves the use of a common **start-rep** procedure to execute a queued transaction. When the procedure is invoked, the local projection queue is examined and all update projections with forkstamps less than that of the transaction to be executed, are applied to the local container.

In summarizing the above discussion on the conservative scheme, we see that the logic and flow of the algorithms are very close to the algorithms used for the kernelized architecture. In both cases the implementation is distributed in nature, and involves the propagation of a WAKE-UP message upwards in security levels.

The proof arguments for demonstrating serial correctness and termination under the kernelized architecture are applicable to the algorithms for the replicated

architecture as well. Hence, for brevity we state the related theorems and omit the proofs.

Theorem 7.1 *The conservative (level-by-level) scheduling algorithms implemented under the replicated architecture maintain the invariant inv-conservative-replicated.*

Corollary 7.1 *The conservative (level-by-level) scheduling implementation for the replicated architecture under invariant inv-conservative-replicated maintains serial correctness.*

Theorem 7.2 *Under the conservative scheduling scheme, all transactions in a session and executing across various containers will eventually terminate and thus guarantee the termination of the user session.*

It now remains to show that the conservative scheme preserves final-state-equivalence. We state and prove this.

Theorem 7.3 *The conservative scheme preserves final-state-equivalence.*

Proof:

By induction on the number of containers, n .

Basis: Consider the basis with $n = 1$. As there is only one container, mutual consistency and final-state equivalence hold trivially.

Induction Step: For the induction hypothesis assume that when n is equal to m , the algorithms preserve final-state equivalence for the m containers. For the inductive step consider $m + 1$ containers with the $m + 1^{\text{th}}$ container C_{m+1} being at level l_{m+1} , and where l_{m+1} is a maximal element in the security lattice and dominates some of the m levels $l_1 \dots l_m$ of the m containers. We consider the levels dominated by l_{m+1}

to belong to the set *dominated* and the remaining levels to belong to the set *not-dominated*. It follows from the induction hypothesis that the container for a level l_k , C_k , (in *dominated*) that is immediately lower than l_{m+1} would have emptied its activation and projection queues and sent a WAKE-UP message to the container at l_{m+1} . This is because if the activation queue were not empty, the projection queue would not be emptied completely, and if the projection queue were not empty, the container C_k would still not have applied the last-updates from lower level transactions (see procedure **wake-up-rep-cons** in figure 7.5). This would make the container at l_k mutually inconsistent with lower containers, contradicting the induction hypothesis. Also, when container C_k has received the WAKE-UP message, it follows that it would have received at that point all the update projections of lower level transactions, originating from containers in the set *dominated*. We now have to consider two cases:

1. Case 1: There are no transactions pending at C_{m+1} , and thus the activation queue is empty. In this case the processing of the WAKE-UP message calls for the projection queue at C_{m+1} to be emptied and applied to the contents at C_{m+1} . When this has been done, container C_{m+1} will be mutually consistent with all containers at the levels in the set *dominated*.
2. Case 2: If there are one or more pending transactions, the activation queue will be dequeued one at a time and started in forkstamp order. Now every start of such a queued transaction will diminish the contents of the projection queue, and thus when the last transaction in the activation queue is started, the projection queue will be empty except for few last-updates from lower level transactions. These remaining update projections will be applied to the contents at C_{m+1} when the last transaction in the activation queue terminates. At this point the contents of container C_{m+1} will be mutually consistent with the containers for the levels in the set *dominated*.

```

Procedure fork-rep-cons(level-parent, level-create, forkstamp,
update-projection)
{
  %Let level-create be the level of the local transaction and container
  Create a new transaction tt at level-create with identifier id;

  %Initialize variables for tt
  tt.id ← id;
  tt.level-parent ← level-parent;
  tt.level-create ← level-create;
  tt.forkstamp ← forkstamp;
  tt.status ← 'non-terminated';

  %This is a priority queue of update projections
  enqueue(projection-queue, update-projection, forkstamp);
  %This is also a priority queue of transactions waiting to be activated
  enqueue(activation-queue, tt);
}
end procedure fork-rep-cons;

```

Figure 7.3: Processing fork requests under conservative scheduling

In either case above, mutual consistency is achieved at container C_{m+1} and this proves that final-state equivalence is preserved across the $m+1^{\text{th}}$ container C_{m+1} and thus across all containers. \square

7.4.2 Aggressive Scheduling Algorithms

We now briefly discuss the implementation of the aggressive scheduling scheme. In addition to the conservative scheme, the implementation of the aggressive scheme requires every container C_j at level j to maintain the following data structure:

Transaction-history: this is a list maintained for each level $i < j$, and maintains for every transaction forked from level i , its id, forkstamp, status and other information.

A transaction history is required to be maintained at every container and keeps track of the forked transactions at dominated levels. It is important to note that this his-

```
Procedure term-rep-cons(level-term, last-update, term-forkstamp,  
last-forkstamp)  
{  
  %Update local container with the last set of updates issued by tt  
  apply the updates in last-update to local container;  
  
  %Post these updates to higher levels  
  For each level > level-term do  
    post-update-rep-cons(level, last-update, increment(last-forkstamp));  
  End-For  
  
  %Dequeue and start the next transaction at level level-term  
  If the activation-queue is not empty  
  then %Let mm be the transaction at the head of the activation queue  
    dequeue(activation-queue, mm);  
    start-rep(mm);  
  else  
    Repeat  
      dequeue (projection-queue, update-projection);  
      apply update-projection to local container;  
    Until projection-queue = empty;  
    Send a WAKE-UP message to containers at immediately higher levels;  
  End-If  
}  
end procedure term-rep-cons;
```

Figure 7.4: Processing terminate requests under conservative scheduling

```

Procedure wake-up-rep-cons
{
  %See if a wake-up message has been received from all immediate lower levels
  If a WAKE-UP message has been received from all immediate lower levels
  then
    %Dequeue and start the next transaction at level level-term
    If the activation-queue is not empty
    then      %Let tt be the transaction at the head of the local activation-
    queue
      dequeue(activation-queue, tt);
      start-rep(tt);
    else
      Repeat
        dequeue (projection-queue, update-projection);
        apply update-projection to local container;
      Until projection-queue = empty;
      Send a WAKE-UP message to containers at immediately higher levels;
    End-If
  End-If
}
end procedure wake-up-rep-cons;

```

Figure 7.5: Processing wake-up requests under conservative scheduling

```

Procedure post-update-rep-cons(local-level, update-projection, forkstamp,
term-flag)
{
  %The propagation-list (update projection) from the lower container is
  unconditionally queued
  enqueue(projection-queue, update-projection, forkstamp);
}
end procedure post-update-rep-cons;

```

Figure 7.6: Processing posted updates with conservative scheduling

```
Procedure start-rep(tt)
{
  %Let tt be the transaction to be started
  counter ← 1;
  Repeat
  %Treat the projection-queue at the level of tt as a list and examine it
  element-wise
    Read (projection-queue[counter], pp);
    If pp.forkstamp ≤ tt.forkstamp
    then
      apply pp to local container;
      delete (projection-queue, pp);
    End-If
    counter ← counter + 1;
  Until counter = length-of(projection-queue);

  %Begin executing tt
  execute(tt);
}
end procedure start-rep;
```

Figure 7.7: Updating the local container before starting a transaction

tory itself is a replicated data structure and snapshot. The need for the maintenance of this history arises from the fact that a container cannot read-down information at lower level containers. Recall that the front-end in the replicated architecture sends messages only in an upwards direction in the lattice. Hence the relevant information has to be gathered with the help of snapshots maintained by constantly sending messages upwards in the lattice. It is the sending of such messages and the maintenance of snapshots such as transaction histories that pose additional complexity in implementing the aggressive scheduling scheme.

The aggressive scheduling algorithm is governed by the following invariant:
Inv-aggressive-replicated: *A transaction is executing at a container at level l only if all non-ancestor transactions (in the corresponding transaction tree) with smaller fork stamps at containers for levels l or lower, have terminated.*

The description of the scheduling algorithms is similar to that of the kernelized architecture, with the difference that we now have to post update projections at the right time to the appropriate containers. A container always looks at its transaction histories for dominated levels to see if the start-up of the next transaction would violate the above invariant. The detailed algorithms are presented in figures 7.8, 7.9, 7.11, and 7.12 and should be self-explanatory.

Proofs

For brevity, we omit the proofs to demonstrate that that our algorithms preserve serial correctness. The arguments are similar to those made for the aggressive scheme under the kernelized architecture. We state the related theorems below for completeness.


```

Procedure fork-rep-agg(level-parent, level-create, forkstamp, update-  

projection)
{
  %Let level-create be the level of the local transaction and container
  Create a new transaction tt at level-create with identifier id;

  %Initialize variables for tt
  tt.id ← id;
  tt.level-parent ← level-parent;
  tt.level-create ← level-create;
  tt.forkstamp ← forkstamp;
  tt.status ← 'non-terminated';

  %Update local transaction history
  append(transaction-historylevel-parent, tt);

  %See if the update projection from the parent can be applied at the local con-  

tainer and if tt can be started immediately
  If  $\forall l \leq \text{level-create}, \neg \exists$  any transaction  $c \in \text{transaction-history}_l$ :
    ( $c.\text{level-create} \leq \text{level-create} \wedge c.\text{forkstamp} < tt.\text{forkstamp}$ 
     $\wedge c.\text{status} = \text{'non-terminated'}$ )
  then
    apply update-projection to local container;
    start-rep(tt);
  else
    %This is a priority queue maintained in forkstamp order
    enqueue(projection-queue, update-projection, forkstamp);
    %This is also a priority queue of transactions waiting to be activated
    enqueue(activation-queue, tt);
  end-if
}
end procedure fork-rep-agg;

```

Figure 7.8: Processing fork requests under aggressive scheduling

```

Procedure wake-up-rep-agg
{
  %Let tt be the transaction at the head of the local activation-queue
  dequeue(activation-queue, tt);
  start-rep(tt);
}
end procedure wake-up-rep-agg;

```

Figure 7.9: Processing wake-up requests under aggressive scheduling

Theorem 7.4 *The aggressive scheduling algorithms implemented under the replicated architecture maintain the invariant inv-aggressive-replicated.*

Corollary 7.2 *The aggressive scheduling implementation for the replicated architecture under invariant inv-aggressive replicated maintains serial correctness.*

Theorem 7.5 *Under the aggressive scheduling scheme, all transactions in a session and executing across various containers will eventually terminate and thus guarantee the termination of the user session.*

We now show that the aggressive scheme preserves final-state-equivalence. We state and prove this as a theorem.

Theorem 7.6 *The aggressive scheme preserves final-state-equivalence.*

Proof:

By induction on the number of possible terminations, n , in a session.

Basis: Consider the basis with $n = 1$. In this case we have only one termination, that of the root transaction. The procedure **term-rep-agg** in figure 7.10 processes terminate requests, and calls for the update projection of the root transaction to be posted to the local container as well as all higher containers. Each higher container,

```

Procedure term-rep-agg(level-term, last-update, term-forkstamp,
last-forkstamp)
{
  % Record the termination of transaction tt at level-term
  For each level  $l < \text{level-term}$  do
    If ( $pp \in \text{transaction-history}_l \wedge pp.\text{forkstamp} = tt.\text{forkstamp}$ )
      then  $tt.\text{status} \leftarrow \text{'terminated'}$ ; End-If End-For

  % Update local container with the last set of updates issued by tt
  apply the updates in last-update to local container;
  % Post these updates to higher levels
  term-flag  $\leftarrow \text{'true'}$ ;
  For each level  $> \text{level-term}$  do
    post-update-rep-agg(level, last-update, increment(last-forkstamp), term-flag, tt);
  End-For

  % See if the update projections for last-updates from lower levels can be applied
  quit-flag  $\leftarrow \text{'false'}$ ;
  For all levels  $l \leq \text{level-term}$  do
    If  $\exists$  any transaction  $q \in \text{transaction-history}_l : (q.\text{status} = \text{'not-terminated'} \wedge$ 
 $q.\text{level-create} \leq \text{level-term})$  then quit-flag  $\leftarrow \text{'true'}$ ; exit for; end-If; end-For;

  If quit-flag =  $\text{'false'}$  then
    Repeat
      dequeue (projection-queue, update-projection);
      apply update-projection to local container;
    Until projection-queue = empty; exit procedure; End-If

  % Check if a queued transaction at level level-term can be started
  % Let mm be the transaction at the head of the activation queue
  If the activation-queue is not empty
  then If  $\forall l, l \leq \text{level-term}, \neg \exists$  any transaction  $c \in \text{transaction-history}_l :$ 
    ( $c.\text{forkstamp} < mm.\text{forkstamp} \wedge c.\text{status} = \text{'not-terminated'}$ )
    then dequeue(activation-queue, mm); start-rep(mm); End-If End-If

  % Check if a transaction at levels  $\geq \text{level-term}$  can be started
  For all levels  $l \leq \text{level-term}$  do
    If  $\exists$  a transaction  $c \in \text{transaction-history}_l$  with  $c.\text{level-create} > \text{level-term} \wedge$ 
 $c.\text{forkstamp} > tt.\text{forkstamp} : \neg \exists$  any non-ancestor transaction  $k$ 
    with  $(\text{level}(k) \leq \text{level}(c) \wedge k.\text{forkstamp} < c.\text{forkstamp} \wedge$ 
     $\text{transaction-history}_{\text{level}(k)}.k.\text{status} = \text{'not-terminated'})$ 
    % We checked to see if c was not preceded by a lower-level active or queued
    % non-ancestor transaction in any of the transaction-histories searched
    then Send a WAKE-UP message to the container at level(c); End-If End-For
  } end procedure term-rep-agg;

```

Figure 7.10: Processing terminate requests under aggressive scheduling

```

Procedure post-update-rep-agg(local-level, update-projection, forkstamp,
term-flag, tt)
{
  %See if the posted update can be applied
  If  $\forall l \leq \text{local-level}$ ,  $\neg \exists$  any transaction  $c \in \text{transaction-history}_l$ :
    ( $c.\text{level-create} \leq \text{local-level} \wedge c.\text{forkstamp} \leq \text{tt.forkstamp}$ 
     $\wedge c.\text{status} = \text{'non-terminated'}$ )
    then
      apply update-projection to local container;
    else
      %This is a priority queue maintained in forkstamp order
      enqueue(projection-queue, update-projection, forkstamp);
  end-if

  If term-flag = 'true'
  then %Record the termination of transaction tt
    transaction-historytt.level-parent.tt.status  $\leftarrow$  'terminated';
  end-if
}
end procedure post-update-rep-agg;

```

Figure 7.11: Processing posted updates

```

Procedure record-new-transaction(transaction, level-parent)
{
  %Update local transaction history for level level-parent
  append(transaction-historylevel-parent, transaction);
}
end procedure record-transaction;

```

Figure 7.12: Recording the fork of computations at lower levels

on receiving the projection, will find that there are no lower level transactions with smaller forkstamps than the terminated root, and apply the update projection from its queue. Each higher container will thus be brought up-to-date with the updates of the root transaction and thus preserving final-state equivalence.

Induction Step: For the induction hypothesis, assume that when n is equal to m , final-state equivalence is guaranteed. For the induction step, let $n = m + 1$. In other words, there are $m + 1$ possible terminations, and given that the first m terminations preserve final-state equivalence, we have to show that the $m + 1^{\text{th}}$ termination preserves final-state equivalence. Consider the transaction t_{m+1} at container C_{m+1} that causes the $m + 1^{\text{th}}$ termination. By the induction hypothesis, we are guaranteed that C_{m+1} will receive all update projections from dominated containers. Some of these projections would be applied to the contents of C_{m+1} as soon as they are received, while others will be queued in the projection queue (as shown in procedure **post-update-rep-agg** of figure 7.11). When t_{m+1} starts, all the queued update projections originating from lower level transactions with smaller forkstamps than t_{m+1} would also be applied to C_{m+1} . Finally when t_{m+1} terminates all remaining update projections will be emptied and applied to C_{m+1} along with its last-updates. This guarantees the mutual consistency of container C_{m+1} with all lower level containers. It now remains to show that mutual consistency is preserved with containers higher than C_{m+1} . This follows from the fact when t_{m+1} terminates, all its update projection would be sent to all higher containers where they would be subsequently applied. Thus final-state equivalence is preserved across all $m + 1$ terminations, and this concludes the proof. \square

Discussion

We conclude this chapter on the replicated architecture with an observation. We have assumed that a transaction is characterized by the property of failure atomicity. It is

thus a unit of recovery. This may be inappropriate for emerging application environments where transactions tend to be of long durations and cooperative in nature, and the amount of work lost due to the abort of an entire transaction, unacceptable. If failure atomicity can be relaxed, we may allow the updates of individual subtransactions to commit (made permanent) independent of the outcome of other subtransactions.

Although it may be possible to relax the failure atomicity property of individual transactions in a session without affecting confidentiality, one wonders if it is feasible to achieve the inverse for an entire user session? In other words, is it possible to maintain atomicity of an entire session without violating confidentiality? That is, all of the component transactions in a session commit or abort without any impact on security (confidentiality). As observed by Mathur and Keefe in [MK93], atomicity and security are conflicting requirements. If a session has component transactions at many levels, we cannot guarantee atomicity without introducing covert channels. At best we can only hope to reduce the bandwidth of such channels. For a discussion of an approach to do this using compensating transactions, the reader is referred to [MK93].

Chapter 8

Inter-session Synchronization

So far we have looked at the issues of concurrency and scheduling within a single user session. We now focus on how objects can be shared across multiple concurrent user sessions. In the database literature, schemes to achieve this generally fall under the category of concurrency control and transaction management. Our purpose here is not to discuss a comprehensive concurrency control scheme, but only to give a basic usable secure solution to object sharing across user sessions. Discussion of a comprehensive transaction model for multilevel systems is beyond the scope of this dissertation. We address in detail inter-session concurrency for the kernelized and replicated architectures, and briefly discuss how some of these algorithms can be utilized in the trusted subject architecture in a secure (confidentiality preserving) manner.

8.1 Inter-session Concurrency for the Kernelized Architecture

Our approach to object sharing is based on a checkin/checkout access data model [LP83]. There exists a public single-version database from which user sessions check-out objects as needed. The objects are checked out into local workspaces (private databases) of individual user sessions. When all activity associated with an object has ceased, the object is checked back into the public database. Due to concurrent

activity in a user session, computations within a user session may view several versions of the same object. However, visibility across user sessions is limited to the public database which maintains only the latest version of every object.

8.1.1 Multilevel Checkin/Checkout of Objects

One of the considerations in designing an object sharing and transaction management scheme is that of formulating and maintaining some notion of inter-session correctness. Conventional database management schemes primarily support transactions that are short-lived and competitive. Interactions and visibility across such transactions are curtailed and the correctness of concurrent transactions is governed by serializability. However, if we examine the applications that are impelling the development of object-oriented database technology, we find that they are characterized by requirements that differ from those utilizing conventional databases. These applications are generally found in environments that call for cooperative work, such as computer-aided design. In such environments, serializability as a correctness criterion needs to be relaxed, and interactions between concurrent transactions have to be promoted rather than curtailed. In light of this, in our further discussions we do not assume that serializability is enforced.

We now discuss how a checkin/checkout scheme can be coupled with the model of r-transactions presented in the last chapter. Our choice of a checkin/checkout scheme as opposed to other conventional schemes directly follows from the above assumption that transactions are cooperative in nature. We provide the following commands to implement a checkin/checkout scheme:

1. **Public-checkout(R/W):** Checks out an object from the public database.
2. **Public-checkin:** Checks in an object into the public database.

3. **Local-checkout(R/W):** Checks out an object from the local workspace of a user session.
4. **Local-checkin:** Checks in an object into a session's local workspace.

The local commands differ from the public ones as their effects are internal to a session, and thus do not affect the visibility or availability of objects to other concurrent user sessions. A checkout operation can be requested in read (R) or write (W) mode. A checkout in W mode is permitted only if the computation generating the requesting subtransaction and the object to be checked out are at the same level. On the other hand, whenever a computation (or more precisely a subtransaction) requests a checkout of a lower level object, the request is granted in read (R) mode only. Multiple subtransactions may checkout the same object (or version of an object) in R mode. However, if a subtransaction first checks out a version in W mode, then no subtransaction at the same or higher levels may check out the version in either R or W mode (as checkouts in R mode conflict with those in W mode). On the other hand, when a high subtransaction first checks out a lower level object in R mode, it is understood that the high subtransaction is given a read-only snapshot of the object at the time. A low subtransaction will be allowed to checkout the same object in W mode before the high subtransaction has checked it in. Thus the checkout of low subtransactions are always given priority. While the checkin operation is necessary for any object checked out in W mode, it is redundant and can be ignored for any object checked out in R mode.

If a requested object has not been checked out by a user session so far, a public checkout request is issued. If however the object had been previously checked out from the public database by the session, it is simply checked out from the session's local workspace. In either case, when the subtransaction terminates, the object is checked

back into the local workspace of the session. A final version of every object that has been updated will eventually be checked back into the public database as explained in the remainder of this section.

When a subtransaction succeeds in checking out a version from a session's local workspace, it is guaranteed that the state of the object so read will never be invalidated in the future. This is because once a version becomes available for checkout in R mode to higher level subtransactions within the same session, we are guaranteed that such a version will never be updated again. To put it another way in transaction processing terminology, a checkout in R mode will always read *committed* values of objects. The implication of this is basically that high level subtransactions cannot develop *abort dependencies* on lower level ones, internally within a session. If such dependencies were possible, then a high level subtransaction would have to abort if a low level subtransaction from which it read, aborts.

As mentioned before, serializability is *not* enforced across user sessions. However, a subtransaction in a session will see only committed states of objects that are updated by other sessions. This is ensured by requiring all public checkin operations from a session to be deferred until the root computation in the session terminates. We consider a session to be logically and semantically committed at the point the root computation terminates normally (i.e., not due to an error or exception). This guarantees that no *abort dependencies* will develop across user sessions. The absence of such dependencies ensures that a session A would not have to abort because another session B from which it read, aborts.

8.1.2 Checkin/Checkout Variations

We now give two variations of a checkin/checkout scheme. They differ basically in how and when objects checked out from the public database are checked back

into the public database, for access by other user sessions. They thus offer different granularities of interactions across user sessions. These variations can be applied to both conservative and aggressive intra-session scheduling strategies. In a *level-by-level checkin/checkout* variation, an object that is updated at a level l by a session, is made visible to another session only when all updates to all objects at level l , by the session, have been completed. In the second *computation-by-computation checkin/checkout* variation, an object is made visible (checked in) as soon as all the subtransactions associated with a computation have terminated.

Level-by-level Checkin/Checkout Schemes

The basic idea is to checkin (commit) objects to the public database, one level at a time. Thus conceptually, we can implement this with processing and propagation of a *level-has-committed* message upwards in the security lattice. With a conservative scheduling scheme, the *level-has-committed* message can be piggybacked onto the *wake-up* message. On the other hand, with aggressive scheduling, the *level-has-committed* message has to be explicitly propagated. We describe both variations below.

The level-by-level checkin/checkout scheme can be combined with the conservative scheduling strategy as follows:

1. A subtransaction checks out the required objects from either its session's local workspace, or from the public database (the latter if any required object has not been previously checked out by the session).
2. When a subtransaction terminates all checked out objects are checked back in to the session's local workspace.
3. If a *wake-up/level-has-committed* message has been received from all immediate

lower levels, and all computations and associated subtransactions at a level say l , have terminated (i.e., when the level manager at l finds its local queue to be empty), then the level manager at l checks in the latest versions of all updated objects into the public database. This is followed by step 4.

4. After all updated objects at level l have been checked into the public database, a *wake-up/level-has-committed* message is sent to all immediate higher levels by the local level manager at level l .

In the conservative scheme above, the receipt of a *wake-up/level-has-committed* message from a lower level is a guarantee that no fork requests will be forthcoming from the lower level. However, in an aggressive level-by-level scheme, this is no longer true. In fact, a level may receive many wake-up messages from a lower level. A *level-has-committed* message can thus no longer be piggybacked onto a *wake-up* message, but rather has to be explicitly propagated, starting with the termination of the root computation. In addition to steps (1) and (2) given above for the conservative scheme, we require the following additional steps to achieve this:

- 3'. When the root computation terminates, we check in all updated objects to the public database and send a *level-has-committed* message to all immediate higher levels.
- 4'. When a *level-has-committed* message has been received from all immediate lower levels, and the local level manager finds its queue to be empty, it checks in all updated objects to the public database. The level manager then propagates the *level-has-committed* message to its immediate higher levels.

Computation-by-computation Checkin/Checkout Schemes

A computation-by-computation checkin/checkout scheme releases (checks in) objects to the public database much earlier in comparison to the level-by-level scheme. Thus on the average, the availability of objects for checkout, across user sessions, is increased as waiting times are reduced. The scheme can be combined again with both conservative and aggressive scheduling. In either case the basic idea is the same. All objects checked out by a computation, or more precisely the set of subtransactions generated by the computation, are checked back into the *public* database as soon as the computation terminates. Contrast this with the level-by-level checkin scheme where we have to wait for all computations at the associated level to terminate. In other words, when the last subtransaction associated with a computation terminates, all checked out objects are checked back into the public database. However for objects checked out in W mode, only the latest version of every object is checked back in. It is obvious that this variation can result in objects being shuffled back and forth from the public database with much greater frequency than the level-by-level scheme.

8.2 Inter-session Concurrency for the Trusted Subject Architecture

The level-by-level and computation-by-computation checkin/checkout schemes can also be implemented for the trusted subject architecture. However, we now have to demonstrate that the concurrency control schemes are secure in that they cannot be exploited for covert channels. In particular, we need assurance that a trusted multilevel subject such as the session manager cannot introduce any interference.

The arguments for a confidentiality proof for our checkin/checkout scheme can be built from the following:

- The checkout requests of low transactions never conflict with higher requests.

This means that the checkout requests of low transactions are never delayed or rejected due to the existence of higher level transactions. Intuitively, we can now establish noninterference by purging the requests of high level transactions and showing that they leave the order and timing of low level requests unaffected within a session, as well as across sessions.

8.3 Inter-session Concurrency for the Replicated Architecture

Having discussed the kernelized and replicated architectures, we now turn our attention to inter-session concurrency control for the replicated architecture. We do not address the issue of concurrency control between sessions at a single container, rather focus on multiple containers. Every container is assumed to provide some local concurrency control.

We assume the following:

- Every container C_j at level j , uses some local concurrency control scheme L_j .
- All containers share a system-low real-time clock. This is a reasonable assumption since the replicated architecture is not for a distributed system, but rather to be implemented on a single (central) machine. The value read from this clock is used to maintain a global serial order for sessions and transactions.

We discuss three approaches to inter-session synchronization and concurrency control that provide increasing degrees of concurrency across user sessions. To elaborate, consider the four sessions S_a , S_b , S_c , and S_d as shown in figure 8.1(a). Sessions S_a and S_b originate at container C_U at level U, while S_c and S_d originate at containers C_C and C_S at levels C and S respectively. The different transactions generated by these sessions are shown in the figure. For example, session S_a generates transactions

T_{a1} at level U, T_{a2} at level C, and T_{a3} at level S. Figures 8.1 (a), 8.1 (b), and 8.1 (c) depict the histories that could be generated by the three inter-session schemes, at the various containers.

In the first scheme, sessions are serialized in a global order that is equivalent to the serialization events of the sessions. If L_j is based on two phase locking, we can use the lock point, which is the last lock step of the root transaction of the session, as its serialization event. If the local concurrency control scheme, L_j is based on timestamping, the timestamp assigned to S_j or the root transaction can be used for the serialization event. In the second approach, this serial order can be successively redefined to interleave incoming newer sessions without affecting the mutual consistency or correctness of the replicas and updates. In the third approach we relax the serial order for the sessions, and instead serialize transactions on a level-by-level basis.

Protocol 1: Globally serial sessions

When a session S_j starts at a container j (i.e., the root transaction executes C_j), the following protocol is observed:

1. S_j makes its resource requests to the local concurrency controller, and its transactions compete with other local sessions that start at C_j .
2. When S_j reaches its serialization event as governed by L_j , the real-time clock is read and its value used to form a *serial-stamp* for S_j .
3. The serial-stamp of S_j is broadcast to all higher level containers.
4. When S_j commits, a *commit-session* message is broadcast to all higher containers. This message may be piggy-backed with the *commit-transaction* message from the root transaction of S_j .

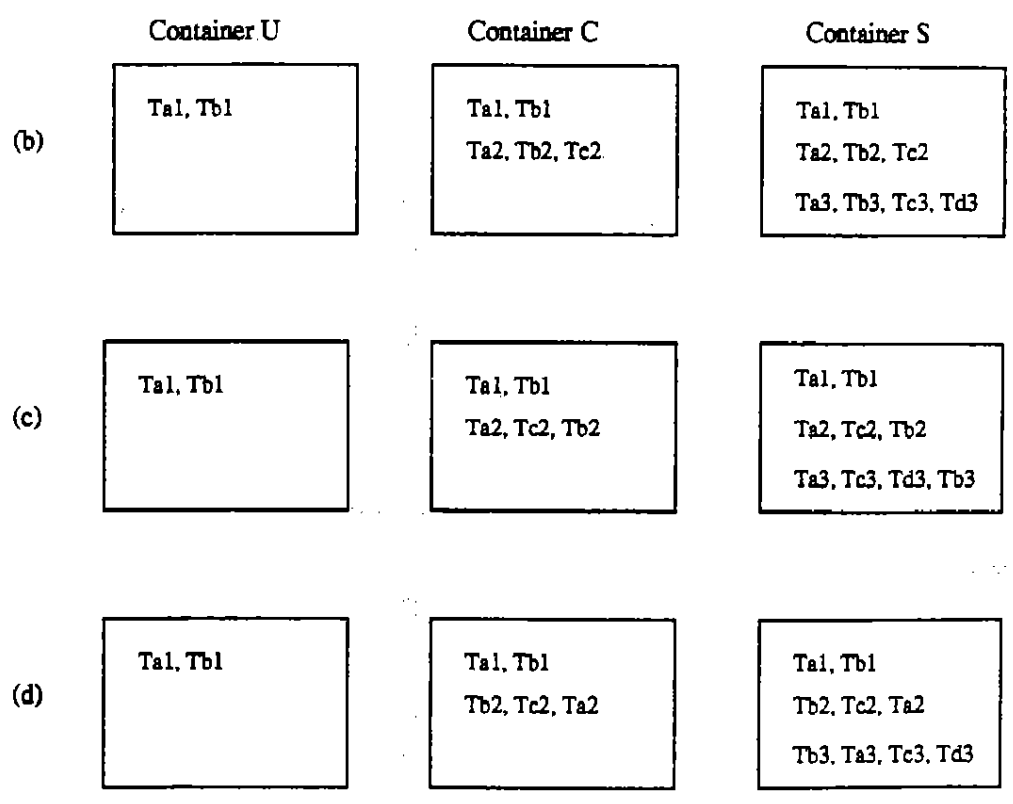
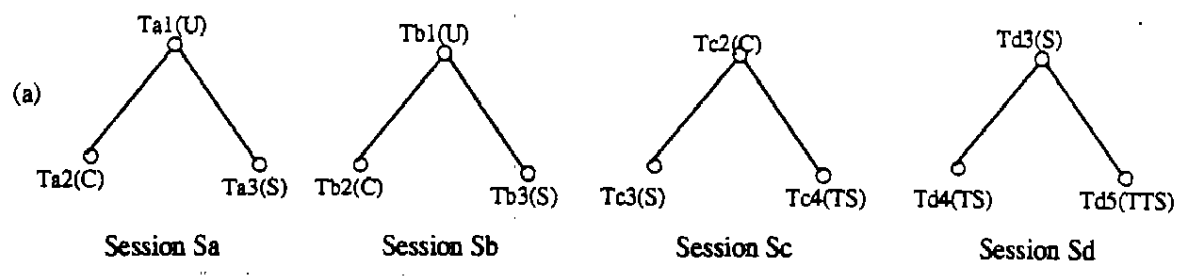


Figure 8.1: Illustrating histories generated with various inter-session synchronization schemes

On receiving the serial-stamp from a container at a lower level, a container, C_k at level k , observes the following rules:

5. All local sessions originating at C_k , and having a smaller serial-stamp than that of S_j , are allowed to commit according to their serial-stamps, and subsequently propagate their updates to containers at levels higher than k .
6. The updates and transactions of S_j are allowed to proceed.
7. All local sessions at C_k having a greater serial-stamp than S_j are allowed to commit only after the *commit-session* notification of S_j is received, and its updates applied as in step 2 above, to C_k .

Several optimizations and variations on the above protocol are possible. It is obvious that the protocol provides minimum concurrency between sessions. In particular, the scheme offers very poor performance if transactions are of long durations. To elaborate, consider what happens if session S_b has sent its serial-stamp to container C but does not commit for a long time. If timestamping is used for the serialization events of sessions at container C , a local session S_c starting at container C_C after the serial-stamp of S_a has been received, will be assigned a greater serial-stamp. Hence, S_c will not be allowed to commit until S_b sends its *commit-transaction* message. The decrease in such concurrency is directly proportional to the size of the window between the serialization and commit events of session S_b .

We can easily improve the performance of the above scheme if S_c were allowed to go ahead and commit even if the *commit-session* message has not been received from S_b . This is possible if S_b has not updated the container C_C at level C so far. We can then re-assign to S_c an earlier serial-stamp than that of S_b . Figure 8.1(b) shows a possible history at the various containers with protocol 1, and figure 8.1(c) shows how

the updates of session S_c can be placed ahead of S_b at container C by giving S_c an earlier serial-stamp than S_b . It is important to note that the relative order between the sessions S_a and S_b is still maintained, but only that S_c is now allowed to come between them. This idea is summarized in protocol 2 below.

Protocol 2: Globally serial sessions with successively redefinable serial-orders

Steps 1 through 6 of Protocol 1 still apply to Protocol 2, but step 7 is modified as below.

When a container C_k receives the serial-stamp from a session S_j at a lower container C_j , the following rules are followed:

- 7'. If there exists a session S_k that has the smallest serial-stamp among the sessions at C_k that have reached their serialization events but not yet committed, and such that S_k has a serial-stamp greater than S_j , then do:
 - (a) If session S_j has not yet updated C_k , then reassign a serial-stamp to S_k that is smaller than the stamp of S_j .
 - (b) Broadcast this new serial-stamp to all higher containers.
 - (c) Allow S_k to update C_k and propagate its updates to higher containers.

The ability of protocols 1 and 2 above, to ensure the mutual consistency of the replicas at the various containers, can be attributed to the way updates are processed. To be more specific, the updates represented in the propagation-lists sent by various sessions, are processed at every container in strict serial-stamp order. A single serial-stamp is associated with the entire set of transactions (updates) that belong to a

session.¹ In other words, a session is the basic unit of concurrency for interleaving updates from multiple sessions. To put it another way, the histories of the updates generated by protocols 1 and 2, guarantee that the individual transactions of two sessions, where each session starts at a different container, cannot be interleaved with each other in any of these histories.

A further improvement to protocol 2 and protocol 1 which we might call protocol 3, can be achieved if the global serial order that is maintained for sessions, is relaxed. Transactions are now serialized in some order on a level-by-level basis. This allows us to exploit more fine-grained concurrency within the structure of a session. The unit of concurrency now is no longer a session, but rather of finer granularity, and thus a transaction. Of course, the key here is exploit such fine-grained concurrency without compromising the mutual consistency of the replicas. The intuition behind this approach is illustrated in figure 8.1(d). Thus we see that the transactions at level U, namely $T_{a,1}$ and $T_{b,1}$ are serialized in the same order at all the containers. However, transactions at level C, namely $T_{a,2}$, $T_{c,2}$, $T_{b,2}$ are serialized in a different order. In particular, the updates from session S_b now come before sessions S_a and S_c . Protocol 2 can easily be modified so that the updates at each level are serialized independently, and made known to the higher containers. Unlike protocols 1 and 2, level-initiator transactions now have to compete with other transactions at the various containers to access data. When an individual transaction reaches its serialization event, the real-time clock is read to form a transaction-serial-stamp, which is subsequently broadcast to higher containers. Mutual consistency of the replicas is achieved by ensuring that updates in the propagation-lists are applied in strict transaction-serial-stamp order.

¹We assume that such associations are kept in some data structure. We also assume that a transaction such as $T_{c,2}$ in figure 8.1(c), running at the container C_C , cannot update the local replicas of data stored at the lower container C_U . Protocols 1 and 2 can guarantee mutual consistency only to the extent that integrity safeguards are available to prevent such events.

We now briefly discuss the correctness of the above protocols. A well known correctness criterion for replicated data is *one-copy serializability* [BHG87]. Protocols 1 and 2 guarantee what one might call *one-copy session serializability*. This gives the illusion that the sessions originating at the different containers execute serially on a one-copy, non-replicated, database. The interactions between transactions as governed by one-copy session serializability is much more restrictive in terms of concurrency and interleaving than one-copy serializability, but implicitly guarantees the latter. The final variation, ie., protocol 3, is less restrictive than the others and does not guarantee one-copy session serializability, but instead maintains one-copy serializability.

Chapter 9

Summary and Conclusions

In this final chapter we summarize the work in this dissertation and highlight future directions for research.

9.1 Research Contributions

This dissertation has focused on the support for RPC-based write-up operations in multilevel secure object-based computing environments. The major complication arises due to the non-primitive nature of such operations. Our solution is novel in that it meets the conflicting goals of secrecy, integrity, and efficiency. We have discussed an asynchronous computational model that calls for concurrent computations to be generated to service RPC-based write-up requests, and a multiversioning approach to synchronizing such concurrent computations. The feasibility of implementing this computational model under three high-assurance multilevel architectures was also demonstrated. These architectures offer varying trade-offs in terms of complexity of implementation.

We end this discussion by listing the specific contributions.

1. **Elaboration of the message filter model.** This dissertation has addressed the architectural and other implementation issues required to map the message filtering functions from an abstract to an executable specification. In doing

this, our effort has demonstrated the feasibility of the message filter model and increased its viability as a practical solution for multilevel secure object-oriented systems.

2. **Feasibility of Three Architectures.** We have demonstrated the feasibility of the message filter model in three popular high-assurance architectures that represent the current focus in security research. These are namely the trusted subject, kernelized, and replicated multilevel system architectures. In particular, the feasibility of a kernelized architecture makes the message filter model attractive for systems that need high assurance.
3. **Complexity of supporting write-up.** We have demonstrated how support of write-up actions although conceptually elegant and simple, is extremely complicated when abstract operations are involved. In particular we have brought to the forefront the trade-offs involved between confidentiality, integrity, and performance.
4. **Concurrency and Synchronization.** The work reported here has provided an original understanding of message-passing, concurrency, and synchronization issues for multilevel secure object-based systems. We have discussed an asynchronous computing model, a multiversioning approach to synchronization, a family of scheduling schemes, as well as a framework and metric for the comparative analysis of these scheduling schemes.
5. **Concurrency control schemes.** We have also briefly discussed various concurrency control schemes for inter-session synchronization. While these schemes are not comprehensive or the most optimal, they do form a basis for further work in multilevel secure object-based concurrency control.

Although our solution is fitting for object-oriented databases and cast in that context, it is important to emphasize that it has wider applicability in any multilevel environment that needs to support write-up operations.

In contrasting our work with other proposals for enforcing mandatory security in multilevel object-oriented systems, we note that while they all address confidentiality, the dimension of integrity is largely ignored. Many of the other solutions assume that the TCB provides protection against signaling channels. But how will the TCB do this? Solutions which are implementation dependent are highly vulnerable to the changes and evolution of computer hardware and performance characteristics. Even if timing channels were closed, without synchronization the integrity problem remains unsolved. We believe it is not easy to coin a complete implementation independent solution without the rigor and detail that we have discussed in this paper.

9.2 Future Work

We now highlight some future directions for research.

Failure and Recovery

Clearly, failure and recovery issues need further investigation. Our approaches do not ensure atomicity of user sessions. If we think of a user session as a transaction, then we have what has now been popularly called a *multilevel transaction*. This is a transaction that reads and writes at multiple security levels. Perhaps the most significant result here is that atomicity and confidentiality are conflicting goals, as observed in [MK93]. If we are to allow for failures and exceptions when computations are running, then guaranteeing atomicity would result in covert channels. More work needs to be done to investigate approaches to reduce such channels in the process of ensuring atomicity.

Beyond RPC Semantics

With synchronous RPC semantics, the intended semantics of concurrent computations is clear and can be used to provide synchronization and to ensure correctness. A future direction would be to go beyond RPC semantics. This would lead to an investigation of more elaborate synchronization, checkpointing, and recovery schemes, given any user supplied notion of correctness.

Concurrency Control

The concurrency control schemes for inter-session synchronization given here are rather rudimentary and need further investigation. While it is always possible to invent another concurrency control scheme, we believe the real challenge would be to develop a comprehensive framework to reason about confidentiality, integrity, and availability of transactions in multilevel environments. In object-based environments we need to reason about atomicity, consistency, isolation, and durability (the so called ACID properties) of transactions differently from traditional transactions. Some preliminary effort in this direction has been undertaken by the author in [TS93].

Client-Server Implementation

Our architectural elaborations have not considered the impact of client-server architectures and technologies. Multilevel security for client-server systems are still an open issue and it is not clear how the general mechanisms for these systems will impact object-based computing.

Discretionary Access Control

Another interesting direction is the investigation of how discretionary access control mechanisms fit into the message filter framework.

Bibliography

Bibliography

- [Agh87] G. Agha. Actors: A conceptual foundation for concurrent object-oriented programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, Cambridge, MA, 1987.
- [BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [BL76] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2997, The Mitre Corp., Burlington Road, Bedford, MA 01730, USA, March 1976.
- [BTMD89] M. Branstad, H. Tajalli, F. Mayer, and D. Dalva. Access Mediation in a Message Passing Kernel. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 66–72. IEEE, May 1989.
- [Cip90] Cipher. Minutes of the First Workshop on Covert Channel Analysis. In *Cipher*, July 1990.
- [CM92] Oliver Costich and John McDermott. A multilevel transaction problem for a multilevel secure database systems and its solution for the replicated architecture. *Proc. IEEE Symposium on Security and Privacy, Oakland, California*, pages 192–203, May 1992.
- [Cos92] Oliver Costich. Transaction processing using an untrusted scheduler in a multilevel secure database with replicated architecture. *Database Security, V: Status and Prospects*, Carl E. Landwehr, ed., North-Holland, Amsterdam, pages 173–189, 1992.
- [Cou83] National Research Council. Multilevel data management security. Technical report, Committee on Multilevel Data Management Security, 1983.

- [CVW+88] T. A. Casey, S. T. Vinter, D. G. Weber, R. Varadarajan, and D. Rosenthal. A Secure Distributed Operating System. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 27-38. IEEE, April 1988.
- [DHP89] K. Dittrich, M. Hartig, and H. Pfefferle. Discretionary access control in structurally object-oriented database systems. *Database Security, II: Status and Prospects*, Carl E. Landwehr, ed., North-Holland, Amsterdam, pages 105-121, 1989.
- [GM82] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. of the IEEE Symposium on Research in Security and Privacy*. IEEE, May 1982.
- [GM84] J.A. Goguen and J. Meseguer. Unwinding and Inference Control. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 75-86. IEEE, May 1984.
- [Hu91] W. M. Hu. Reducing timing channels with fuzzy time. In *Proc. of the IEEE Symposium on Research in Security and Privacy*. IEEE, May 1991.
- [JK90] S. Jajodia and B. Kogan. Integrating an Object-Oriented Data Model with Multilevel Security. In *Proc. of the IEEE Symposium on Research in Security and Privacy*. IEEE, May 1990.
- [JS91] S. Jajodia and R.S. Sandhu. A Novel Decomposition of Multilevel Relations into Single-level Relations. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 300-313. IEEE, May 1991.
- [Kee90] T. F. Keefe. *Multilevel Secure Database Management Systems*. PhD thesis, University of Minnesota, September 1990.
- [KJ90] Boris Kogan and Sushil Jajodia. Concurrency control in multilevel-secure databases using replicated architecture. *Proc. ACM SIGMOD Int'l. Conf. on Management of Data, Atlantic City, New Jersey*, pages 153-162, May 1990.
- [KTT88] T. F. Keefe, W. T. Tsai, and M. B. Thuraisingham. A multilevel security model for object-oriented system. *Proc. 11th National Computer Security Conference*, pages 1-9, October 1988.
- [L+77] Barbara Liskov et al. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564-576, August 1977.

- [LP83] R. Lorie and W. Plouffe. Complex objects and their use in design transactions. In *Databases for Engineering Applications*, pages 115–121. ACM, May 1983.
- [Lun90] T. F. Lunt. Multilevel security for object-oriented database system. *Database Security, III: Status and Prospects*, David L. Spooner and Carl Landwehr, eds. North-Holland, Amsterdam, pages 199–209, 1990.
- [LZ74] B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGPLAN notices*, 21(11), 1974.
- [MA91] B. Maimone and R. Allen. Methods for resolving the security vs. integrity conflict. In *Proc. of the fourth RADC Database Security Workshop*, April 1991.
- [Mai89] David Maier. Why isn't there an object-oriented data model. *Proc. of the 11th IFIP World Computer Conference*, pages 793–798, September 1989.
- [McC90] D. McCullough. A Hookup Theorem for Multilevel Security. In *Transactions on Software Engineering*. IEEE, June 1990.
- [MH89] J. A. McDermid and E. S. Hocking. Security Policies for Integrated Project Support Environments. In *Proc. of the IFIP Workshop on Database Security*, September 1989.
- [MJS91] John McDermott, Sushil Jajodia, and Ravi Sandhu. A single-level scheduler for replicated architecture for multilevel secure databases. *Proc. 7th Annual Computer Security Applications Conf., San Antonio, Texas*, pages 2–11, December 1991.
- [MK93] A. G. Mathur and T.F. Keefe. The concurrency control and recovery problem for multilevel update transactions in MLS systems. In *Proc. of the Computer Security Foundations Workshop*. IEEE, June 1993.
- [ML92] J. K. Millen and T. F. Lunt. Security for object-oriented database systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 260–272, May 1992.
- [MO87] M. Mizuno and A. E. Oldehoeft. Information Flow Control in a Distributed Object-Oriented System with Statically Bound Object Variables. In *Proc. of the 10th National Computer Security Conference*, pages 56–67, September 1987.

- [San93] R. Sandhu. On the four definitions of data integrity. *Proc. of the 7th IFIP WG 11.3 Workshop on Database Security*, September 1993.
- [STJ91] R.S. Sandhu, R. Thomas, and S. Jajodia. A secure kernelized architecture for multilevel object-oriented databases. *Proc. of the IEEE Computer Security Foundations Workshop IV*, 1991.
- [STJ92] R.S. Sandhu, R. Thomas, and S. Jajodia. Supporting timing channel free computations in multilevel secure object-oriented databases. In C.E. Landwehr and S. Jajodia, editors, *Database Security V: Status and Prospects*, pages 297-314. North-Holland, 1992.
- [TC89] M. B. Thuraisingham and F. Chase. An object-oriented approach for designing secure software systems. In *Cipher Newsletter of the Technical Committee on Security and Privacy*, pages 7-14, 1989.
- [Thu89a] M. B. Thuraisingham. Mandatory security in object-oriented database system. In *Proc. Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 203-210, October 1989.
- [Thu89b] M. B. Thuraisingham. A multilevel secure object-oriented data model. In *Proc. 12th National Computer Security Conference*, pages 579-590, October 1989.
- [TS92] R.K. Thomas and R.S. Sandhu. Implementing the message-filter object-oriented security model without trusted subjects. In C.E. Landwehr, editor, *Database Security VI: Status and Prospects*. North-Holland, 1992.
- [TS93] R.K. Thomas and R.S. Sandhu. Towards a unified framework and theory for reasoning about security and correctness of transactions in multilevel databases. In T.F. Keefe and C.E. Landwehr, editors, *Database Security VII: Status and Prospects*. North-Holland, 1993.
- [TS94a] R.K. Thomas and R.S. Sandhu. A kernelized architecture for multilevel secure object-oriented databases supporting write-up. *Journal of Computer Security*, 2(3), 1994.
- [TS94b] R.K. Thomas and R.S. Sandhu. Supporting object-based high-assurance write-up in multilevel databases for the replicated architecture. *To appear in the Proceedings of the European Symposium on Research in Computer Security (ESORICS-94)*, 1994.

- [Weg87] P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 393–415. MIT Press, Cambridge, MA, 1987.
- [Weg91] P. Wegner. Design issues for object-based concurrency. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-based Concurrent Programming*, pages 245–256. Springer-Verlag, New York, 1991.

Vita

Roshan Thomas was born on December 9, 1964 in the town of Changanacherry, Kerala State, India. At the age of six, his family moved to Nigeria, on the West African coast. He received the HSC O level certificate from Kings College, Lagos, and the Bachelor of Science in Computer Science from the University of Lagos, Nigeria. He subsequently pursued graduate work in the United States, and obtained the Master of Science degree from the University of Houston, Texas. For the past five years he has pursued doctoral studies at George Mason University, Fairfax, Virginia. His research interests include computer and information security, object-oriented computing, and distributed systems. His other non-academic interests include poetry, philosophy of religion, studies of mythology, classical piano, photography, and tennis.

This dissertation was typeset with \LaTeX^{\dagger} by the author.

[†] \LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's \TeX Program.