The rapid growth of the networking and information-processing industries has led to the development of distributed database management system prototypes and commercial distributed database management systems. In such a system, the database is stored in several computers which are interconnected by some communication media. The aim of a distributed database management system (DDBMS) is to process and communicate data in an efficient and cost-effective manner. It has been recognized that such distributed systems are vital for the efficient processing required in military as well as commercial applications. For many of these applications it is especially important that the DDBMS should provide multilevel security. For example, the DDBMS should allow users who are cleared at different security levels access to the database consisting of data at a variety of sensitivity levels without compromising security. In this paper we discuss multilevel security issues for a DDBMS. We first describe a system architecture, security policy, and data/metadata distribution issues for a multilevel secure DDBMS (MLS/DDBMS). Next we describe issues on query processing and transaction management based on the system architecture considered.

Keywords: Multilevel secure distributed database management systems, System architecture, Security policy, Multilevel data/metadata distribution, Query processing, Transaction management, Concurrency control, Recovery.

1. Introduction

The rapid growth of the networking and information-processing industries has led to the development of distributed database management system prototypes and commercial distributed database management systems [3]. In such a system, the database is stored in several computers which are interconnected by some communication media. The aim of a distributed database management system (DDBMS) is to process and communicate data in an efficient and cost-effective manner. It has been recognized that such distributed systems are vital for the efficient processing required in military as well as commercial applications. For many of these applications, it is especially important that the DDBMS should operate in a secure manner. For example, the DDBMS should allow users who are cleared at different levels access to the database consisting of data at a variety of sensitivity levels without compromising security.

A considerable amount of work has been carried out in providing multilevel user/data handling capability in centralized database management systems, known as multilevel secure database management systems (MLS/DBMS) [1, 5, 8, 9, 14, 18]. In contrast, multilevel secure distributed database management systems (MLS/DDBMS) have
received very little attention.\footnote{An MLS/DBMS is also called a trusted database management system (TDBMS). Similarly, an MLS/DDBMS is also called a trusted distributed database management system (TDDDBMS).} Note that in some DDBMSs limited forms of discretionary security controls (that is, where users access data based on authorizations) do exist [3].

In an earlier article published in this journal [1], we described our preliminary investigation on multilevel security issues for a distributed database management system. The architecture that we considered was the front-end/back-end type illustrated in Fig. 1. This architecture was chosen simply because it was the one example of a distributed architecture considered in the Air Force Summer Study [1]. In this architecture, a front-end machine is connected to one or more back-end database systems. All requests to the database systems are via the front-end machine. That is, the front-end machine controls the execution of all transactions. As a result the back-end machines cannot execute local applications. This feature is not strictly in accordance with the standard definition of a DDBMS given in Ceri and Pelagatti [3], in which it was stated that:

"A distributed database is a collection of data which is distributed over different computers of a computer network. Each site of the network has autonomous processing capability and can perform local applications. Each site also participates in the execution of at least one global application, which requires accessing data at several sites using a communication subsystem. A distributed database management system supports the creation and maintenance of a distributed database."

In contrast to Fig. 1, Fig. 2 illustrates an architecture that conforms to this definition of a DDBMS. It is this architecture that is being considered in many DDBMS prototypes and products [12]. In this architecture, the nodes are connected via a communication subsystem. This subsystem could be any network such as a local area network or a long-haul network. Each node has its own local DBMS which is capable of handling the local applications. In addition, each node is also involved in at least one global application. In other words, there is no centralized control in this architecture.

In this paper multilevel security issues for a DDBMS based on such an architecture are described. It is primarily these issues that have to be considered in the design of any MLS/DDBMS. Each of the issues considered in this paper is briefly introduced below.

(1) System architecture: we will illustrate a system architecture for an MLS/DDBMS which has been derived from our choice of architecture for a DDBMS (given in Fig. 2). All of the security issues to be described in this paper will be based on this system architecture.
(2) Security policy: we will discuss some issues for a mandatory security policy for an MLS/DDBMS.

(3) Multilevel data/metadata distribution: there are two types of data distribution schemes that should be considered in an MLS/DDBMS. One is the distribution of multilevel data within a node and the other is the distribution of multilevel data across nodes. Both distribution schemes will be discussed. Metadata describes the data in the distributed database. Metadata could also be assigned different security levels. The issues involved in secure metadata management are the design of the metadata and the distribution of the metadata.

(4) Query processing: query processing is one of the more important functions of an MLS/DDBMS. The techniques should ensure that users obtain the data at or below their security level. The data distribution issues should be transparent to the user. An architecture for a distributed query processor as well as strategies for secure query processing will be discussed.

(5) Transaction management: a transaction in a multilevel environment executes at a particular security level. The issues involved in transaction management in an MLS/DDBMS are secure concurrency control and recovery. An architecture for a distributed transaction manager as well as secure concurrency control techniques will be discussed.

The organization of this paper is as follows: section 2 will describe the system architecture that we have considered for an MLS/DDBMS. Mandatory security policy issues will be discussed in section 3. Multilevel data/metadata distribution issues will be discussed in section 4. Query processing in an MLS/DDBMS will be discussed in section 5. Transaction management in a multilevel environment will be discussed in section 6. The paper will be concluded in section 7.

We assume that the reader is familiar with concepts in DDBMS and MLS/DBMS. An excellent discussion on DDBMS concepts is given in Ceri and Pelagatti [3]. A useful starting point for concepts in MLS/DBMS is the Air Force Summer Study Report [1].

2. System Architecture

In an MLS/DDBMS, users cleared at different security levels access and share a distributed database consisting of data at different security levels without violating security. The system architecture for an MLS/DDBMS that we consider in our investigation is shown in Fig. 3. This architecture
B. Thuraisingham/Multilevel Security in Database Management Systems

Fig. 3. System architecture for a MLS/DDBMS.

has been derived from our choice of architecture for a DDBMS (given in Fig. 2). All of the security issues to be described in this paper will be based on this system architecture.

In this architecture, the MLS/DDBMS consists of several nodes that are interconnected by a multilevel secure network. We assume that the nodes are homogeneous. That is, all of the nodes are designed identically. Each node is capable of handling multilevel data. Each node has an MLS/DBMS which manages the local multilevel database. Each node also has a distributed processing component called the secure distributed processor (SDP). The components of SDP, shown in Fig. 4, are the distributed metadata manager (DMM), distributed query processor (DQP), distributed transaction manager (DTM), and the distributed constraint processor (DCP). DMM manages the global metadata. The global metadata include information on the schemata which describe the relations in the distributed database, the way the relations are fragmented, and the locations of the fragments. DQP is responsible for distributed query processing. DTM is responsible for distributed transaction management. DCP is responsible for handling security constraints during query and update processing.

The DQP and DTM communicate with the DMM for the metadata required to carry out their functions. In our design of the SDP we do not have a separate module for update processing. We assume that individual update requests are handled by the DQP. Update requests specified as part of a transaction are handled by the DTM. Since a transaction is a series of query and update requests, we assume that the DQP is invoked by the DTM in order to carry out the individual requests. Two DMMs (or DQPs, DTMs) at different nodes communicate in accordance with the security policy enforced. SDP may be implemented as a set of processes separate from the local MLS/DBMS.

2A multilevel secure network is also called a trusted network. Issues are discussed in Trusted Network Interpretation [17].
3This paper will discuss mainly the issues involved in designing the SDP.
4DQP and DTM will be described in sections 5 and 6 respectively. DMM will be briefly described in section 4. DCP will be described in a forthcoming paper.
5We assume that the database is relational. Multilevel data distribution issues will be discussed in section 4.
6Note that information pertaining to the local database is managed by the local MLS/DBMS.
7Security constraints are rules which assign security levels to the data. For a discussion we refer to Dwyer et al. [6].
8The update operation performed by the DQP is straightforward. A user at level L updates data at level L. In our design of the DQP we have only concentrated on the query operation.
3. Security Policy

The security policy for a computing system consists of a set of policies for mandatory security, discretionary security, integrity and authentication, among others. Our focus is on a mandatory security policy for an MLS/DDBMS. An effective mandatory security policy for an MLS/DDBMS should ensure that users only acquire the information at or below their level.\(^\text{9}\) The basic mandatory security policy for the MLS/DDBMS that we have considered has the following properties.

1. Subjects are the active entities (such as processes) and objects are the passive entities (such as tuples or relations).

2. Subjects and objects are assigned security levels. The set of security levels forms a partially ordered lattice (e.g., unclassified < confidential < secret < top secret).

3. A subject has read access to an object if the subject's security level dominates the security level of the object.

4. A subject has write access to an object if the subject's security level is the security level of the object. (Note that this is a restricted version of the *-property of the Bell and LaPadula security policy [2].)

5. A subject S1 can send a message to another subject S2 if the security level of S2 dominates (i.e. greater than or equal to) the level of S1.

In designing a secure system, it may be necessary for additional security policy extensions to be enforced. Such policy extensions are carried out by trusted subjects.\(^\text{10}\) That is, it has to be ensured that such a subject's security critical functions are

---

\(^\text{9}\) By a security policy we will mean a mandatory security policy.

\(^\text{10}\) See the design of the MLS/DBMS given in Stachour and Thuraisingham [14] for a discussion on security policy extensions.
B. Thuraisingham/Multilevel Security in Database Management Systems

The security architecture that we consider is shown in Fig. 5. We assume that each node has a trusted computing base (TCB). The TCB is the part of the host that enforces the mandatory security policy at that host. The network TCB is responsible for enforcing the network security policy. The TCB hosts various trusted applications such as an MLS/DBMS and an SDP. Additional security policy extensions may be enforced by these applications depending on their designs. In our design, the system must ensure that two DMMs (DQPs, DTMs, DCPs) at different nodes can communicate with each other only if they both operate at the same level. Also, additional security policy extensions are enforced by certain modules of the SDP.

Multilevel Data/Metadata Distribution

The functions of the SDP will depend on the way the data are distributed. In this section we describe multilevel data/metadata distribution issues. We first discuss data-modelling issues and then describe how data and metadata may be distributed within and across sites.

4.1. Local Data Distribution

We assume that the database is based on the relational model at the global and local levels. Therefore, the data model used to represent the database at the global and local levels is a multilevel relational data model. In this section we define a multilevel relational data model that is used to represent the multilevel database at each local node. A multilevel relational data model is the rel-

Note that in our discussion of security issues for a DDBMS, we do not refer to the level of assurance provided by the system. The levels of assurance provided by computing systems are discussed in the Trusted Computer Systems Evaluation Criteria [15]. An interpretation of the TCSEC for multilevel database systems is given in Trusted Database Interpretation [16]. It appears that criteria for evaluating MLS/DDBMSs will not be available in the near future. The assurance provided by the MLS/DDBMS will depend on the assurance provided by the local hosts and the network. The assurance provided by the network is determined by the interpretation of the TCSEC for networks given in Trusted Network Interpretation [17].

4. Multilevel Data/Metadata Distribution

The functions of the SDP will depend on the way the data are distributed. In this section we describe multilevel data/metadata distribution issues. We first discuss data-modelling issues and then describe how data and metadata may be distributed within and across sites.

4.1. Local Data Distribution

We assume that the database is based on the relational model at the global and local levels. Therefore, the data model used to represent the database at the global and local levels is a multilevel relational data model. In this section we define a multilevel relational data model that is used to represent the multilevel database at each local node. A multilevel relational data model is the rel-

For example, the security policies of the local operating system, local MLS/DBMS, SDP, and the network have to be integrated in order to obtain a security policy for the MLS/DDBMS. It should be noted that very little work has been done on integrating the security policies of different modules.

The relational data model was initially developed by Codd [4].
A multilevel relational database consists of a set of multilevel relations. We define a multilevel relation to be a relation in which each tuple is assigned a security level. Figure 6 shows a multilevel relation EMP. Note that the tuples in EMP are either secret or unclassified. Furthermore, if SS # is taken to be the primary key of EMP, then we see that there are two tuples with the same primary key at different security levels. This feature is known as polyinstantiation [14]. In other words, we assume that SS # together with the security level forms the primary key for the relation EMP.

Recombination is the process of forming a view of a relation at a particular security level. The recombination operation at a security level L will involve all tuples of the relation dominated by L. When polyinstantiation is present, a user can request the lower-level polyinstantiated tuples to be removed from his view. If so, only the tuple associated with the maximum security level that is dominated by the security level at which the recombination operation is being performed is considered. Figure 7 illustrates the various views of the multilevel relation EMP that the unclassified and secret users have.

At the physical level, the multilevel-relation EMP could be stored in single-level fragments. Figure 8
B. Thuraisingham/Multilevel Security in Database Management Systems

illustrates this feature. The fragment EMP-U stores the unclassified tuples of EMP and the fragment EMP-S stores the secret tuples of EMP.

We assume that the security level of the schema of a relation is the security level of the user who creates the schema. For every security level \( L \) that dominates the security level of the schema of a relation \( R \), there could be tuples of \( R \) at level \( L \). Therefore, corresponding to an unclassified schema of relation \( R \), there could be tuples of \( R \) at the unclassified, confidential, secret, and top secret levels. Since a multilevel relation can be decomposed into single-level relations, corresponding to every security level \( L \) that dominates the security level of the schema of a relation \( R \), there could be fragments of \( R \) at level \( L \). A user can read any tuple whose security level is dominated by his level. When a user enters a tuple, the tuple is assigned the level of the user and is stored in a fragment of the relation at the security level of the user. If the user's security level is dominated by the security level of the schema of the relation, then the tuple is not entered.

### 4.2. Distribution Across Sites

As stated before, we assume that the global data model is the same multilevel relational model used to represent the local databases. The global multilevel relations could be totally or partially replicated across sites. Furthermore, the relations could also be horizontally fragmented. That is, the global relation is partitioned into horizontal subsets. The subsets could be stored across several sites. Tuples could be polyinstantiated within as well as across sites. A multilevel distributed database stored at two sites is illustrated in Fig. 9. Figure 10 illustrates the global views that the unclassified and secret users have of the database.

---

**Fig. 9.** Multilevel distributed database.

**Fig. 10.** Global views.
4.3. A Note on Metadata Management
Metadata at the global level are managed by the distributed metadata manager (DMM).\textsuperscript{13} The metadata at the global level include information on the schemata of the global relations, the way the relations are fragmented, the allocation of the fragments, and the various constraints enforced. We assume that the metadata are replicated at each node. Furthermore, we assume that metadata are stored at multiple security levels. Metadata stored at level L are also classified at level L. There is a manager at each security level. The managers need not be trusted in the distributed architecture that we have considered.\textsuperscript{14} The manager at level L manages the metadata at level L. The manager at a security level L in node 1 communicates with the manager at security level L in node 2 in order to access remote metadata. Such a scheme is illustrated in Fig. 11.

5. Distributed Query Processing
In section 5.1 we describe an architecture for the DQP and in section 5.2 we discuss strategies.

5.1. Distributed Query Processor
The module that is responsible for processing queries is the distributed query processor (DQP).\textsuperscript{15} The components of the DQP, shown in Fig. 12, are the user interface manager (UIM), query transformer (QT), the query optimizer (QO), and the distributed execution monitor (DEM). The user’s query is parsed by the UIM. The QT transforms the parsed query at the logical level. That is, the query transformation process transforms a global query into equivalent fragment queries. This process is performed according to transformation rules and does not depend on the allocation of the fragments. Discretionary access checks are also performed by the QT. The transformed query is then processed by the QO, which determines the most efficient way to execute the query. That is, the alternate execution strategies produced by the QT must be examined by the QO in terms of the cost of executing them. The cost is usually determined by the number of tuples that are transmitted and the number of operations that have to be performed. The DEM monitors the execution of the query. The DEM interfaces to the local MLS/DBMS.\textsuperscript{16} The metadata required for distributed

\textsuperscript{13}Note that the DQP is part of the SDP described in section 2.
\textsuperscript{14}Note that we are concerned with secrecy issues only. That is, we do not address integrity issues in this paper.
\textsuperscript{15}Note that the DQP is part of the SDP described in section 2.
\textsuperscript{16}In a heterogeneous environment, there will be a local execution monitor (LEM) which will be responsible for the necessary transformation between the global representation and local representation. The DEM will have an interface to LEM and LEM will have an interface to the local MLS/DBMS.
query processing are obtained from the distributed metadata manager (DMM). Communication with a remote DQP is achieved via the Network Interface Manager.

The DQP could be implemented using the process model or the client-server model. In the process model, there is a DQP process for each user. This process executes at the same level as the user. In the client-server model there is a set of DQP processes per security level. A DQP process executing at level L services requests of users at level L. The system must ensure that two DQPs at different nodes communicate only if they both operate at the same level. The DQP operating at level L communicates with the DMM process also operating at level L in order to obtain the necessary metadata. A user’s request at level L will be processed entirely at level L. As a result, information which is not dominated by level L will not be used during the processing. Therefore, no additional security policy extensions are enforced by the DQP. That is, the DQP need not be a trusted process.

5.2. Strategies
In this section, we describe two query-processing strategies for the join operation.\textsuperscript{17} We consider the join operation as it is one of the most time-consuming operations and it has been studied extensively in the past for non-multilevel DDBMSs [3]. We describe the strategies for a join operation involving only two relations.

Let EMP and DEPT be two relations. EMP has attributes SS\#, Name, Salary and D\# with SS\# as the primary key. DEPT has attributes D\#, Dname, and Mgr with D\# as the primary key. We assume that tuples have different security levels. A tuple at level L is stored in a fragment of the relation at level L. That is, the secret tuples are stored in a secret fragment and the unclassified tuples are stored in an unclassified fragment. The fragments could also be partitioned horizontally. Further-

\textsuperscript{17}We have made several optimizations to the strategies presented here. The optimized strategies will be described elsewhere.
more, tuples could also be polyinstantiated. Suppose the query is to form the join between EMP and DEPT over the D# values. Let us also assume that the user requests the lower level polyinstantiated tuples not to be involved in the join operation. We call such a join the restricted join. We assume that polyinstantiation within a node is handled by the local MLS/DBMS. Therefore, we are only concerned with the polyinstantiated tuples across sites.

In the first strategy, called the nondistributed join, the fragments of each relation at or below the user's level are merged first, then the lower-level polyinstantiated tuples are eliminated, and finally the join operation is performed. A possible sequence of operations for the nondistributed join is as follows:

1. Find the sites A and B which have the most number of tuples of the relations EMP and DEPT, respectively, provided the tuples are at or below the user's level.

2. Send all EMP fragments at or below the user's level to site A and all the DEPT fragments at or below the user's level to site B.

3. Form the union of the EMP fragments and DEPT fragments. Subsequently eliminate the lower-level polyinstantiated tuples. Let the resulting EMP and DEPT relations be EMP* and DEPT* respectively.

4. Project the relation EMP* on D#. Send the D# values to node B.

5. Reduce the relation DEPT* using the D# values received. That is, perform the join between the D# values received and the relation DEPT*. Let DEPT** be the reduced relation.

6. Send DEPT** to node A. Perform the join between EMP* and DEPT**. Send the result to the site at which the query was posed.

Figure 13(a) shows a multilevel distributed database stored at two sites. Site 1 has the unclassified EMP fragment EMP1-U and the secret DEPT fragment DEPT1-S. Site 2 has the secret EMP fragment EMP2-S and the unclassified DEPT fragment DEPT2-U. Suppose a secret user requests a restricted join between EMP and DEPT. We will describe how this join operation will be carried out using the strategy described above. Step (1) of the strategy will select site 1 as the site for the merge operation for both relations. Step (2) of the strategy will send both the fragments at site 2 to site 1. Figure 13(b) illustrates the relations EMP* and DEPT* obtained as a result of step (3). Since both relations are at site 1, steps (4) and (5) are not necessary. Figure 13(c) shows the result of the join operation which is obtained as a result of step (6).

In the second strategy, called the distributed join, the join operations are performed between various fragments. Finally, the results of the individual join operations are merged. A possible sequence of operations for the distributed join is as follows:

1. Determine whether it is less costly to send (1) all the DEPT fragments at or below the user's level to each site with an EMP fragment at or below the user's level or (2) all the EMP fragments at or below the user's level to each site with a DEPT fragment at or below the user's level. Let us assume that it is less costly to do (1).

2. Send each DEPT fragment at or below the user's level to any site which has an EMP fragment at or below the user's level.

3. When the DEPT fragments are received at a site, perform the union of these fragments and then eliminate the lower-level polyinstantiated tuples. Let the result be DEPT*.

4. Perform the join operation between DEPT* and the fragment of EMP at or below the user's level at that site. If there are more than two fragments of EMP at or below the user's level, these fragments are merged first before performing the
Fig. 13. Example illustrating strategy 1.
join operation. Note that lower-level polyinstantiated tuples are removed during the merge operation. Let $R_1, R_2, \ldots, R_n$ be the result of the join operation at the various sites. For each SS# values occurring in $R_i (1 \leq i \leq n)$, attach its security level also. This security level is the level of the corresponding EMP tuple used in the join operation.

(5) Transmit all the relations $R_1, R_2, \ldots, R_n$ to a selected site. This site is determined by computing the costs to transmit the tuples to the various sites and selecting the one with the least cost.

(6) Merge $R_1, R_2, \ldots, R_n$. Let the result be $R$. Eliminate from $R$ all lower-level tuples with polyinstantiated SS# values. The result is sent to the site at which the query was posed.

Figure 14(a) shows the same distributed database of Fig. 13(a). Suppose a secret user requests a restricted join between EMP and DEPT. We will describe how the join operation will be carried out using the second strategy described above. In step (1) it will be determined that the DEPT fragments should be transmitted. During step (2) DEPT2-U will be sent to site 1 and DEPT1-S will be sent to site 2. The result of step (3) which produces the relation DEPT* is shown in Fig. 14(b). The result of step (4) which produces the relations $R_1$ and $R_2$ at site 1 and site 2 is shown in Fig. 14(c). Since $R_2$ has fewer tuples, $R_2$ is sent to site 1 as a result of step (5). The final result which is produced as a result of step (6) is shown in Fig. 14(d).

6. Transaction Management

In section 6.1 we provide an overview of the security impact on distributed transaction management. Concurrency control issues are described in section 6.2.

6.1. Distributed Transaction Management

Transaction management in a DDBMS involves the handling of distributed transactions. By a distributed transaction we mean a transaction which executes at multiple sites. The portion of the transaction which executes at a particular site is a subtransaction associated with that site. In a multilevel environment, a distributed transaction executes at the level of the user who requested its execution. Each distributed transaction can execute at one or more nodes. The portion of the transaction that executes at a particular node is a subtransaction and it executes at the same level as the distributed transaction. At each node that the distributed transaction executes, there is an application agent. The agents operate at the same level as that of the transaction. The agents of the same transaction communicate with each other. One of the agents is called the root agent. When a “Begin Transaction” command is encountered, the root agent invokes the distributed transaction manager (DTM) at the same node. This DTM, which acts as the coordinator, operates at the same level as the agent. The coordinator then issues the appropriate “Local Begin” command to its local transaction manager (LTM) and also communicates with other DTMs at the nodes in which the transaction executes in order to inform them of the “Begin Transaction” command. The DTMs at the other nodes communicate with their LTMs. Note that all of the DTMs as well as the LTMs operate at the same level as that of the transaction. When a “Commit” command is encountered, then the coordinator carried out the commit protocol. Figure 15 illustrates a model for distributed transaction management in a multilevel environment.

The security policy for the distributed transaction management extends the security policy for local transaction management. We assume the following policy for a centralized system:

20Note that the DTM is part of the SDP described in section 2.
21The LTM is part of the local MLS/DBMS.
22Although not shown in the figure, we assume that the DTM at level L communicates with the DMM at level L in order to obtain the necessary metadata to execute the transaction. Also, the DTM invokes the DQP in order to process the individual query and update requests associated with the transaction.
Fig. 14. Example illustrating strategy 2.
Each transaction is executed at the level of the user who requests the execution. This level must dominate the level assigned to the program itself. (Note that a transaction is a program and could be assigned a security level.)

- A transaction does not change levels during its execution.

- A transaction reads from and writes into objects according to the mandatory security policy enforced by the system.

Extensions to distributed transaction management are the following:

- A distributed transaction executes at the level of the user who requested the corresponding application to be executed.

- A distributed transaction's subtransactions also execute at the same level.

- The subtransactions execute in accordance with the security policy enforced by the local system (it is assumed that all nodes enforce the same policy).

- A distributed transaction (at the global level) reads and writes objects in accordance with the global mandatory security policy enforced (we assume that this is the same as for the local systems).

- A distributed transaction does not change levels during execution.

- Two DTMs at different nodes communicate only if they both operate at the same security level.

The two-phase commitment protocol used for a DDBMS can be extended for an MLS/DDBMS. In this protocol, one site (usually the site of origin) acts as the coordinator and other sides where the transaction executes act as participants. The coordinator as well as the participants maintain log records and write all their actions on the records. First the coordinator sends a message to the participants to prepare to commit. Each participant sends a message to the coordinator as to whether it is ready to commit. If all of the participants have answered positively, then the coordinator sends a commit message to all participants. If otherwise, the coordinator sends an abort message to the pa-
participants. If a participant receives a commit message, then the subtransaction at that site is committed. If an abort message is received, the subtransaction is aborted. In a multilevel environment, since the entire transaction executes at the same security level, the coordinator as well as the participants operate at the same security level during the entire duration of the transaction.

Recovery techniques should ensure that transactions are recovered in the presence of system and network failures. In order to do this, appropriate log files are maintained. Several recovery protocols, such as extensions to the two-phase commit protocols, have been proposed for non-multilevel systems. The security impact on these protocols is yet to be investigated.

If transactions are executed serially, then there will be a performance bottleneck. Therefore transactions usually execute concurrently as specified by a schedule. The component that is responsible for scheduling transactions is called the scheduler. The local scheduler is responsible for schedules in a local MLS/DBMS and the global scheduler is responsible for schedules at the global level. The schedules are such that it must be ensured that the multiple transactions maintain the consistency of the distributed database. The techniques that ensure this consistency are called concurrency control techniques. They ensure that a schedule is serializable. That is, it is equivalent to a serial schedule. The following local and global serializability conditions will guarantee that the schedule is serializable in a distributed environment.

A local serializability condition for a schedule S is as follows [3]:

There is a serial schedule S* such that for each pair of conflicting operations O_i and O_j, if O_i precedes O_j in S, then O_i precedes O_j in S*. (Note that O_i and O_j are conflicting operations if they request operations on the same data item and at least one of the two operations is a write operation.)

A global serializability condition is as follows [3]:

If T_1, T_2, ..., T_n are transactions executing concurrently, and if S_1, S_2, ..., S_m are the local schedules of these transactions, then there exists a total ordering of transactions T_1, T_2, ..., T_n such that for each pair of conflicting operations (that is, two different transactions request to access a data object and one of the requests is a write request) O_i and O_j from transactions T_i and T_j respectively, O_i precedes O_j in any schedule S_i, S_2, ..., S_m if and only if T_i precedes T_j in the total ordering. In order for the global schedules to be serializable, it is necessary for the local schedules to be serializable.

In a multilevel environment, in addition to consistency, it has to be ensured that high-level transactions cannot affect the lower-level ones at the local as well as the global levels. Concurrency control techniques must ensure that consistency as well as security has to be preserved. In our approach, there is a scheduler per security level. The scheduler at level L is responsible for scheduling transactions at level L. The concurrency control algorithms should ensure that the actions of higher-level transactions cannot affect the lower-level ones. At the same time the consistency of the distributed database must also be preserved. Concurrency control algorithms are discussed in the next section.

6.2. Concurrency Control

In this section we investigate the applicability of two concurrency control techniques for distributed transactions in a multilevel environment. These techniques are locking [7] and time stamping [13]. To simplify the discussion we assume that there are only two security levels; Unclassified and Secret. The techniques can actually handle any number of security levels. We have adapted other concurrency control techniques such as validation [10] to a multilevel environment. These techniques will be described in a future paper.
6.2.1. Locking
In the locking technique, two-phase locking is used to provide concurrency control (i.e., a transaction acquires all necessary locks first before releasing any). Two types of locks may be obtained. They are shared locks and exclusive locks. A shared lock for a data item is requested by a transaction for the read operation. The request is granted if no other transaction has a write lock for the same data item. An exclusive lock is requested by a transaction for the write operation. The request is granted when no other transaction has either a shared lock or an exclusive lock for the same data item.

In the locking approach to providing concurrency control, there is a potential for covert channels if subjects at different security levels access the same object concurrently. For example, suppose a secret transaction requests read access to an unclassified data object Q while an unclassified transaction requests either read or write access to Q. If both transactions want only to read Q, then they can both get shared locks for Q at the same time. That is, there is no possibility for a channel. However, if the unclassified transaction wants to write, then there is a possibility for a channel. For example, a secret transaction can issue a sequence of requests of the form

\[(\text{get read lock } Q, \text{ release read lock } Q),\]
\[(\text{get read lock } Q, \text{ release read lock } Q), \ldots\]

The unclassified transaction could issue requests to write into Q. If the secret transaction already has a read lock, then the request issued by the unclassified transaction will be denied. The unclassified transaction could detect the pattern of the denials and acceptances that it gets by requesting a series of write requests to the object Q. If both transactions collude, then secret information can be covertly passed to the unclassified transaction. A similar problem occurs when a secret transaction requests read or write access to a secret object Q while an unclassified transaction requests write access to Q.

Below we investigate a possible solution to adapting the locking approach when multilevel transactions are executing concurrently. To simplify the discussion we assume that there are two security levels: secret and unclassified. Since a secret data object can be accessed only by secret transactions, we are concerned with only unclassified data objects. We assume that there are two copies (both unclassified) of an unclassified data object: one copy for unclassified transactions and one for secret transactions. For a data object X, let X₁ be the copy for unclassified transactions and X₂ be the copy for secret transactions. When unclassified transactions request locks, the actions of the secret transactions have no affect. The unclassified transactions contend for locks on X₁. As soon as an unclassified transaction finishes updating X₁, a new copy of X₁ is created. Let this copy be X₃. X₁ will then be the new version of X for the secret transactions. After an unclassified transaction requests the write lock for X₁, any read request on X by a secret transaction would be directed to X₃. X₂ may be deleted later by a garbage collector. To ensure consistency, X₂ should be deleted only if there are no locks queued for it. Since we assume that X₃ is an unclassified object, the process which deletes X₂ must be trusted.

What we actually mean is that secret information is covertly passed to the subject which executes the unclassified transaction. If the garbage collector is untrusted, then it must be unclassified. It will have no way of knowing whether the read requests of the secret transactions are completed. Sufficient time could be given before deleting X₁. However, there can be no guarantee that all the read requests on X₁ are completed. A second solution would be to create the copy X₃ for secret transactions at the secret level. Then a secret process can delete X₁ after all read requests are completed. However, in this case, if there are N security levels which dominate the unclassified level, then a copy of the unclassified data object X needs to be maintained at each of the N levels. There must also be a way to create the copies at these higher levels.
We show that the serializability condition is satisfied assuming that locking is two-phased. Let X and Y be two data objects. Let \( T_i \) be an unclassified transaction and \( T_j \) be a secret one. Let \( T_i \) and \( T_j \) have conflicting operations \( O_i \) and \( O_j \), respectively on X and Y for a schedule \( S \). That is, \( T_i \) requests the operation \( O_i \) on X and Y and \( T_j \) requests the operation \( O_j \) on X and Y. Since \( T_i \) is unclassified and \( T_j \) is secret, \( O_i \) is a write operation and \( O_j \) is a read operation. We show that if \( O_i \) precedes \( O_j \) (\( O_j \) precedes \( O_i \)) for X, then \( O_i \) precedes \( O_j \) (\( O_j \) precedes \( O_i \)) for Y. Since the two-phase locking is used, when \( O_i \) requests a write lock on X it also requests a write lock on Y. When \( T_j \) requests a read lock on X, it also requests a read lock on Y. There are two cases to consider:

1. Let \( T_j \) request a read lock on X and Y before \( T_i \) requests a write lock on X and Y.

Since \( T_j \) requested locks on X and Y before \( T_i \) requested locks on X and Y, even if the operation \( O_i \) is actually carried out first on either X or Y, \( O_j \) will be performed on the older versions of X and Y and not on the new versions of X and Y created as a result of the operation \( O_i \). The end result would be as if \( O_j \) precedes \( O_i \) for both X and Y.

2. Let \( T_i \) request a write lock on X and Y before \( T_j \) requests a read lock on X and Y.

If \( T_i \) requests a write lock on X and Y first, then the read lock requested by \( T_j \) will be queued until \( O_i \) is completed and the new versions of X and Y are created. \( O_j \) will be performed on the new versions. The end result would be as if \( O_i \) precedes \( O_j \) for both X and Y.

We have shown that in both cases the schedule \( S \) is equivalent to a serial schedule. That is, the serializability condition is satisfied. Next, we discuss distributed two-phase locking in a multilevel environment.

In distributed two-phase locking, each distributed transaction acquires all the locks before releasing any. Let X and Y be unclassified data items at sites 1 and 2 respectively. Let \( T_i \) and \( T_j \) be two distributed transactions which request conflicting operations \( O_i \) and \( O_j \), respectively, on X and Y. Let \( S_1 \) and \( S_2 \) be the schedules at sites 1 and 2 respectively. We show that if \( O_i \) precedes \( O_j \) (\( O_j \) precedes \( O_i \)) in \( S_1 \) then \( O_i \) precedes \( O_j \) (\( O_j \) precedes \( O_i \)) in \( S_2 \). We consider two cases:

1. Let \( T_j \) request a read lock on X and Y before \( T_i \) requests a write lock on X and Y.

Since \( T_j \) requested locks on X and Y before \( T_i \) requested a lock on X and Y, even if the operation \( O_i \) is actually carried out first in either \( S_1 \) or \( S_2 \), \( O_j \) will be performed on the older versions of X and Y and not on the new versions of X and Y created as a result of the operation \( O_i \). The end result would be as if \( O_j \) precedes \( O_i \) in both \( S_1 \) and \( S_2 \).

2. Let \( T_i \) request a write lock on X and Y before \( T_j \) requests a read lock on X and Y.

If \( T_i \) requests a write lock on X and Y first, then the read lock requested by \( T_j \) will be queued until \( O_i \) is completed and the new version of X and Y are created. \( O_j \) will be performed on the new versions. The end result would be as if \( O_i \) precedes \( O_j \) in both \( S_1 \) and \( S_2 \).

6.2.2. Time Stamping

In the time-stamping approach for centralized systems, all transactions are given a time stamp when they begin. The values of the time stamps increase with time. Each data object has a read stamp and a write stamp. The read stamp of an object is the time stamp of the transaction which last read from it. A write stamp of an object is the time stamp of the transaction which last wrote into it. A transaction's read request to an object is permitted only if its time stamp is greater than the write stamp of the object. A transaction's write request to an object is permitted only if its time stamp is greater than both the read and write stamps of the object. We first discuss how the time-stamping technique could be adapted for a multilevel centralized sys-
We then discuss global serializability condition for the time-stamp technique in a multilevel environment.

Our objective is to ensure that higher-level transactions do not interfere with lower-level ones. In addition, the integrity of the database has to be maintained also. We propose the following modification to the time-stamp algorithm. As in the case of the locking approach, we use two copies of an unclassified data object: one for unclassified transactions and one for secret transactions. Whenever a secret transaction issues a read request to an unclassified data object X, first ensure that all unclassified transactions with lower transaction numbers have completed their requests on X. Then process the read request issued by the secret transaction. Note also that after an unclassified transaction finishes updating an object, a new copy of the object is made for the secret transactions. The old copy used by the secret transaction may be deleted by a garbage collector. To ensure consistency, this copy must be deleted only after all secret transactions with transaction numbers lower than that of the unclassified transaction have finished the read operation on it. This means that the process which deletes it must be trusted.26

We now show that the serializability condition is satisfied. Let Ti be an unclassified transaction and Tj be a secret one. Let Oi and Oj be the conflicting operations requested by Ti and Tj, respectively, on X and Y. Note that Oi is a write operation and Oj is a read operation. We have the following cases:

(1) The time stamp of Ti is less than the time stamp of Tj. In this case, Oi precedes Oj for both X and Y.

(2) The time stamp of Ti is greater than the time stamp of Tj. Then we have the following:

(a) Let Oj precede Oi for X. If Oj precedes Oi for Y then the serializability condition is satisfied. If Oj does not precede Oi for Y, then the only way this is possible is for Oj to be requested before Oi is requested, but Oj is carried out before Oi is carried out. However, Oj will be carried out on the older version of the data object and not on the version that results after Oi completes. The end result is as if Oj precedes Oi for Y.

(b) Let Oj precede Oi for X. The only way this is possible is for Oj to be requested before Oi is requested, but Oj is carried out before Oi is carried out. However, Oj will be carried out on the older version of the data object and not on the version that results after Oi completes. The end result is as if Oj precedes Oi for X. If Oj precedes Oi for Y, then the serializability condition is preserved. If Oj precedes Oi for Y, as before, we can argue that the end result is as if Oj precedes Oi for Y.

Next, we discuss distributed time-stamp management in a multilevel environment. To satisfy the global condition, suppose Oi and Oj are conflicting operations performed by Ti and Tj, respectively, on X and Y. Let Tj precede Ti. We need to show that Oi precedes Oj in each local schedule. Let Ti be unclassified and Tj be secret. Then Oi is a write operation and Oj is a read operation. Let S1 and S2 be the schedules at sites 1 and 2 respectively. We have two cases to consider:

(1) The time stamp of Ti is less than the time stamp of Tj. In this case, Oi precedes Oj in both S1 and S2.

26Note that the global time stamps are assigned in a distributed environment. For a discussion on assigning global time stamps we refer to Ceri and Pelagatti [3]. Also, in a multilevel environment, it must be ensured that there is no information leakage from a higher level to a lower level.
(2) The time stamp of $T_i$ is greater than the time stamp of $T_j$. Then we have the following:

(a) Let $O_j$ precede $O_i$ in $S_1$. If $O_j$ precedes $O_i$ in $S_2$, then the serializability condition is satisfied. If $O_j$ does not precede $O_i$ in $S_2$, then the only way this is possible is for $O_j$ to be requested before $O_i$ is requested, but $O_j$ is carried out before $O_i$ is carried out. However, $O_j$ will be carried out on the older version of the data object and not on the version that results after $O_i$ completes. The end result is as if $O_j$ precedes $O_i$ in $S_2$.

(b) Let $O_i$ precede $O_j$ in $S_1$. The only way this is possible is for $O_j$ to be requested before $O_i$ is requested, but $O_i$ is carried out before $O_j$ is carried out. However, $O_j$ will be carried out on the older version of the data object and not on the version that results after $O_i$ completes. The end result is as if $O_j$ precedes $O_i$ in $S_2$. If $O_j$ precedes $O_i$ for $Y$, then the serializability condition is preserved. If $O_j$ precedes $O_i$ in $S_1$, as before, we can argue that the end result is as if $O_j$ precedes $O_i$ in $S_2$.

7. Conclusion

For many applications, it is important to provide secure, efficient and reliable access to multilevel databases stored at different sites. Present distributed database systems do not provide adequate mechanisms to meet this objective. Hence the solution requires the operation of a distributed database management system (DDBMS) capable of handling multilevel data. Such a system is also called a multilevel secure distributed database management system (MLS/DDBMS).

The purpose of this paper is to identify the impact of multilevel security on the functions of a DDBMS. We first defined an architecture for an MLS/DDBMS and discussed a security policy and multilevel data distribution issues. We then described various issues on query processing and transaction management. Our current work on MLS/DDBMS includes (1) developing a prototype for secure distributed query processing and (2) conducting research and simulation studies for secure transaction management. In addition, we are also investigating (1) issues on processing security constraints in a distributed environment, (2) network security issues for an MLS/DDBMS, and (3) security issues for heterogeneous database systems. This work, which will be reported in forthcoming papers, will eventually lead toward the design and development of operational MLS/DDBMSs for various applications.

Acknowledgment

We gratefully acknowledge the Department of the Navy (SPAWAR) for supporting our work on security issues for distributed database management systems under contract F19628-89-C-0001. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Navy or the United States Government.

References


Bhavani Thuraisingham is a lead engineer with the MITRE Corporation working in database security. Her recent research contributions include security in distributed database management systems, multilevel object-oriented data models, logic for secure data/knowledge base systems, and the complexity of the inference problem. She is also leading team efforts on the design and implementation of a secure distributed query processor, database inference controller, security constraint processor, secure multimedia/object-oriented database system, and secure transaction management algorithms. Her current research interests are in database security, object-oriented programming, AI techniques, distributed processing, and mathematical logic in computer science. Previously, she was at Honeywell Inc. where her work included the design of the secure database system Lock Data Views, and before that she was at Control Data Corporation where she was involved with the development of the computer network CDCNET. She was also an adjunct professor of computer science and a member of the graduate faculty at the University of Minnesota. Dr. Thuraisingham received the MS degree in computer science from the University of Minnesota, MSc degree in mathematic logic from the University of Bristol, and the PhD degree in recursive functions and computability theory from the University of Wales, Swansea. She has published over a hundred technical papers and reports including over thirty journal articles in database security, distributed processing, artificial intelligence, and mathematical logic. Dr. Thuraisingham is a member of the British Computer Society, Association of Symbolic Logic, IEEE Computer Society, and the ACM, and has served on the program committee of the IEEE Local Computer Networking Conference. She is also a member of the Editorial Board of the Journal of Computer Security and the editor of the Third RADC Database Security Workshop Proceedings.