Chapter 4

Transaction Management in a Mobile Data Access System

K. SEGUN, A.R. HURSON, V. DESAI

Computer Science and Engineering Department
The Pennsylvania State University
University Park, PA 16802

A. SPINK

School of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802

L.L. MILLER

Computer Science Department
Iowa State University
Ames, IA 50011

Abstract Advances in wireless networking technology and portable computing devices have led to the emergence of the mobile computing paradigm. As a result, the traditional notion of timely and reliable access to global information sources in a distributed system or multidatabase system is rapidly changing. Users have become much more demanding in that they desire and sometimes even require access to information anytime, anywhere. The amount and the diversity of information that is accessible to a user are also growing at an exponential rate. Compounding the access to information is the wide variety of technologies with differing memory, network, power, and display requirements. Within the scope of distributed databases

and multidatabases, the issue of concurrency control as a means to provide timely and reliable access to the information sources has been studied in detail. In an MDAS environment, concurrent execution of transaction is more problematic due to the power and resource limitations of computing devices and lower communication rate of the wireless communication medium. This article is intended to address the issue of concurrency control within a multidatabase and an MDAS environment. Similarities and differences of multidatabases and MDAS environment are discussed. Transaction processing and concurrency control issues are analyzed. Finally, a taxonomy for concurrency control algorithms for both multidatabases and MDAS environments is introduced.

Keywords: Multidatabase, Mobile computing environment, Mobile data access system, concurrency control, transaction processing.

4.1 Introduction

The need to maintain and manage a large amount of data efficiently has been the driving force for the emergence of database technology and more recently E-commerce. Initially, data was stored and managed centrally. As organizations became decentralized, the number of locations, and thus local databases, increased. The need for shared access to multiple databases was inevitable. Geographical distribution of data, demand for highly available systems and autonomy coupled with economical issues, availability of low cost computers, advances in distributed computing, and the demands of supply chain based E-commerce, are among the pressing issues behind the transition towards distributed database technology.

Design of distributed database management systems (DBMS) has had an impact on issues such as concurrency control, query processing/optimization and reliability. Traditionally, distributed DBMSs have been built in a top down fashion – building separate databases and distributing data among them [12]. This approach has the advantage that fixed standards can be set before the databases are built, simplifying the issues of data distribution, query processing, concurrency control and reliability. The local DBMSs are typically homogeneous with respect to the data model implemented and present the same functional interfaces at all levels. The global system has control over local data and processing. Solutions developed in a centralized environment can be typically extended to fit this model, resulting in a tightly coupled global information-sharing environment. This approach to designing a distributed database system is possible only if the design process is started from scratch. The issue is how to effectively distribute the data, having knowledge of resources like machine capacity, network overhead, and semantics of data.

The natural extension to the distributed database system came in the form of applications of distributed systems in integrating preexisting databases to make the most use of the data available. Most organizations already have their major databases in place. It would be impractical to move this data into a common database, since it would not only be expensive, but the independence of managing individual databases also would be lost. The alternative is logical integration of

data, so as to provide a view of one logical database. This can be viewed as a bottom up approach to distributed database design. The databases themselves are loosely coupled and could potentially differ in data models, and transaction and query processing schemes used - heterogeneous databases or multidatabases [10]. A key feature is the autonomy that individual databases retain to serve their existing customer set. The goal of integrating the databases is to provide users with a uniform access pattern to data in several databases, without modifying the underlying databases and without requiring knowledge of the location or characteristics of the various DBMSs. Solutions from a centralized environment cannot be directly applied in such an environment, autonomy and heterogeneity being restricting factors. Unlike the distributed approach, this could be viewed as being motivated by efficient integration of data instead of efficient distribution of data.

Multidatabase systems (MDBS) are not simply distributed implementations of centralized databases, but can be seen as much broader entities that present their own unique characteristics. This in turn raises several interesting research issues over and above those for centralized databases. Some of these issues, like query optimization and transaction management, are rooted in a centralized environment. Others, such as data distribution, have their roots in distributed databases. However, issues such as local autonomy are unique to the multidatabase problem.

An important emerging computing paradigm is mobile computing. Thanks to recent advances in computer and telecommunications technology, and the subsequent merging of both technologies, mobile computing, particularly as an important technological infrastructure for E- commerce is now a reality. A Mobile Data Access System (MDAS) is a multidatabase system that is capable of accessing a large amount of data over a wireless medium. Such a system is realized by superimposing a wireless mobile computing environment on a multidatabase system [38]. Mobility raises a number of additional challenges for multidatabase system design. Current designs are not capable of resolving the difficulties that arise as a result of the inherent limitations of mobile computing: frequent disconnection, high error rates, low bandwidth, high bandwidth variability, limited computational power, and limited battery life. Wireless transmission media across wide-area telecommunication networks are an important element in the technological infrastructure of E-commerce [62]. Effective development of guided and wireless-media networks will enhance delivery of World Wide Web functionality over the Internet. Using mobile technologies will enable the purchase of E-commerce goods and services anywhere and anytime

Multiple users, in general, could access a database, which implies multiple transactions occurring simultaneously. This is especially true in a distributed database system where various users at different sites could access independent databases. In data processing applications (e.g. banking, stock exchange) the need for reducing access times and maintaining availability, reliability, and integrity of data is essential. Effective E-commerce is based on the successful development and implementation of multidatabase systems. E-commerce based businesses need effective solutions to the integration of Internet front-end systems with diverse data in legacy- distributed

databases. A key challenge for E-commerce is the need for real-time concurrent access to distributed databases containing accounting, marketing, inventory, sales, production systems, and vendor information.

Concurrent access to data is a natural way to increase throughput and reduce response time. Database operations require extensive I/O operations and, in addition, a distributed environment has to cope with delays in the network. These characteristics motivate interleaving the execution of several transactions. Concurrent transaction processing raises the possibility of interference. It is safe to concurrently access data items as long as they are independent, but in case of related data items, accesses should be coordinated – concurrency control being the activity of coordinating concurrent accesses to shared data [6]. In a multidatabase environment, transactions could span over multiple databases. Concurrency control in such an environment should not only synchronize subtransactions of a transaction at the respective sites, but also the transactions as a whole at the global level. In addition, the coordination of transactions in a multidatabase environment should enforce minimal changes to the local databases, with autonomy of the component databases being a distinctive feature.

In the MDAS environment, transactions tend to be long-lived. This is due to frequent disconnection, the limited bandwidth constraints experienced by the mobile user, and the mobility of users. The communication path tends to increase as users move from one administrative domain to another even if physical distances traversed are short. Concurrency control in such an environment should take into account the effects of disconnection, limited bandwidth, mobility and portability. Concurrency control must strive to reduce the communication, reduce computation, and conserve the battery life of the mobile unit.

Computer software and hardware are failure prone. Incomplete transactions due to failure can lead to inconsistencies. Failures can even lead to loss of data. This stresses the need for a database to have effective recovery mechanisms or methods to maintain atomicity of transactions. In addition, in a distributed case, failure of some sites should not halt the execution of the whole system, availability being an important consideration. To handle these issues, proper transaction management schemes should be incorporated into a database system.

It is the responsibility of transaction management schemes to ensure correctness under all circumstances. By correctness, a transaction should satisfy the following ACID properties [31]:

- **Atomicity:** Either all operations of a transaction happen or none happen. State changes in a transaction are atomic.

- **Consistency:** A transaction produces results consistent with integrity requirements of the database.

- **Isolation:** In spite of the concurrent execution of transactions, each transaction believes it is executing in isolation. Intermediate results of a transaction should be hidden from other concurrently executing transactions.

- **Durability:** On successful completion of a transaction, the effects of the transaction should survive failures.

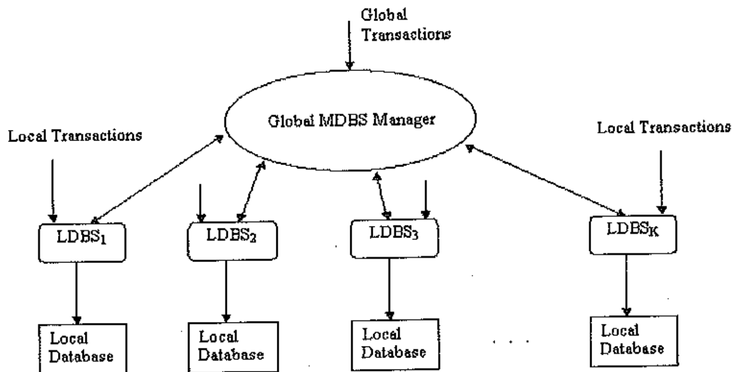
Extensive research has been done to maintain the ACID properties of transactions in centralized and tightly coupled distributed database environments [31]. The emergence of the need to access preexisting databases, as in a multidatabase environment, imposes newer constraints and difficulties in maintaining the ACID properties. The difficulties stem essentially from the requirement of maintaining the autonomy of the local databases. This implies that the local sites have full control over the data at their respective sites. The consequence is that the local executions are outside the control of the global multidatabase system. Transaction processing in such a fully autonomous environment can give rise to large delays, frequent or unnecessary aborts, possible inconsistencies on failure and hidden deadlocks, just to name a few problems that are encountered. Inevitably, certain assumptions and tradeoffs have to be made, usually compromising the autonomy of the local databases, in order to maintain the goals of transaction processing in general.

In the next section, we provide a brief introduction to multidatabase and MDAS systems and their research issues. In Section 4.3, concurrency control and transaction processing issues in MDBS and MDAS are discussed and their differences from traditional distributed systems are addressed. Section 4.4 looks at the existing solutions for transaction management in both environments. Section 4.5 addresses application based and advanced transaction management. Section 4.6 discusses our experiments with the V-locking algorithm in an MDAS environment. Finally, Section 4.7 concludes this article.

4.2 Multidatabase Characteristics

A multidatabase system is a distributed system that acts as a front end to multiple local DBMSs. It provides a structured global system layer on top of existing local DBMSs. The global layer is responsible for providing full database functionality and interacts with the local DBMSs at their external user interface. The end user gets an illusion of a logically integrated database, hiding intricacies of different local DBMSs at the hardware and software levels. Thus, a multidatabase can be viewed as a database system formed by independent databases joined together with a goal of providing uniform access to the local DBMSs. A multidatabase system, in general, can be represented by the architecture shown in Figure 4.1.

The primary objective of the multidatabase is to place as few restrictions on the local DBMSs as possible. Another goal (or maybe more of a consequence) of forming a multidatabase system is the recognition of the need for certain basic standards in the development of databases so as to simplify global information sharing in the future. Heterogeneity is a term commonly used in a multidatabase environment. In general, heterogeneity can occur due to differences in hardware, operating systems, data models, communication protocols, to name a few. In a multidatabase environment, to make global information sharing a reality, heterogeneities in data models,



LDBS: Local Database Management System
MDBS: Multidatabase System

Figure 4.1. Multidatabase System.

schema, query languages, query processing, and transaction management schemes have to be resolved.

4.2.1 Taxonomy of Global Information Sharing Systems

There are a wide range of solutions for global information systems in a distributed environment, with terms like distributed databases, federated databases, and multidatabases being among the most commonly used. The distinction arises from the degree of autonomy and the manner in which the global system integrates with the local DBMSs. A tightly coupled system means global functions have access to low-level internal functions of the local DBMSs. In a loosely coupled system, the local DBMS allows global control through external user interfaces only. The amount of control that a local DBMS retains over data at its site after joining the global system is the basis for the following taxonomy.

- **Distributed Databases:** A distributed database is the most tightly coupled global information sharing system. The global manager has control over transactions occurring both globally and locally. Such systems are typically designed in a top down fashion, with global and local functions implemented simultaneously. Logically, distributed databases give the view of centralized databases, with data at multiple sites instead of a single one.

- **Federated Database:** Federated database systems are more loosely coupled than distributed database systems. The participating DBMSs have significantly more control of data at their respective sites. In a federated database system, each of the local DBMSs decides what part of the local data is shared.

They cooperate with other local DBMSs in the federation for global operations. The DBMSs in the federation have typically been designed in a bottom-up manner, but when they join the federation, they give up a certain amount of local freedom.

- **Multidatabase Systems:** Multidatabase systems are the most loosely coupled systems. Here, the local DBMS retains full control over the local data even after joining the global system. In a multidatabase system, it is the responsibility of the global system to extract information and resolve various aspects of heterogeneity for global processing.

This classification highlights independence and autonomy of the local databases as two important features of a multidatabase system. The relationship between multidatabases and autonomy merits more attention and is highlighted in the following discussion.

4.2.2 MDBS and Node Autonomy

Node autonomy is one of the key concepts in a distributed system [27]. A MDBS is a distributed system formed to allow uniform access to multiple local DBMSs, wherein local operations have priority and may be more frequent than global operations; thus, enforcing autonomy of the underlying sites becomes important. On the other hand, the local site could contain legacy databases; transforming this data to suit global needs would be too expensive. Thus, economical constraints are also motivating factors in preserving local autonomy. Issues such as isolating some local data from global access magnify the need for site autonomy. This allows the local DBA to restrict the information available to the global user. Autonomy may be a suitable feature from the global standpoint as well; in case of failure, it helps check the effects of a local failure from propagating throughout the system. Autonomy could come in different forms [27]:

- **Design Autonomy:** The local DBMS should not be made to change its software or hardware platform to join a multidatabase system. In short, the local DBMS should remain as is on becoming a part of a global system. The global software can be looked upon as an add-on to the existing system. The primary reason for design autonomy is economics - an organization may have significant capital invested in existing hardware, software, and user training. This is especially relevant for systems designed in a bottom up manner. Heterogeneity arises in distributed systems if they are allowed to retain their design autonomy.

- **Communication Autonomy:** The local DBMS has the freedom to decide what information it is willing to share globally. This can imply that a local DBMS may not inform the global user about transactions occurring locally, which makes the task of coordinating global transaction execution spanning over multiple sites extremely difficult. Synchronization of global transactions

is not the responsibility of the local DBMS. Local databases in federated and distributed database systems do not retain their communication autonomy since they provide information for global transaction coordination [10].

- **Execution Autonomy:** A local DBMS can execute transactions submitted at the local site in any manner it desires. This implies that global requirements for the execution of a transaction at a local site may not be honored by the local DBMS. The local DBMS has the freedom to unilaterally abort any transaction executing at that local site. This is due to the fact that the local DBMS does not treat the global transaction as a special transaction; it is like any other local transaction executing at that node.

In the next subsection, we examine some of the issues that arise due to autonomy and the inherent heterogeneity in the multidatabase environment.

4.2.3 Issues in Multidatabase Systems

The primary issues that are heavily influenced by local database autonomy are outlined in the following:

- **Schema Integration:** The local databases have their own schema; the goal here is to create an integrated schema to give a logical view of an integrated database. Schema integration is difficult when component databases differ in name, format and structure. Briefly, naming differences occur due to difference in naming conventions, wherein semantically equivalent data could be named differently or semantically conflicting data could be named the same. Format differences include differences in data types, domain, scale, precision, and item combinations, whereas structural differences occur due to differences in data structures. Schema integration has been discussed extensively in [35,39].

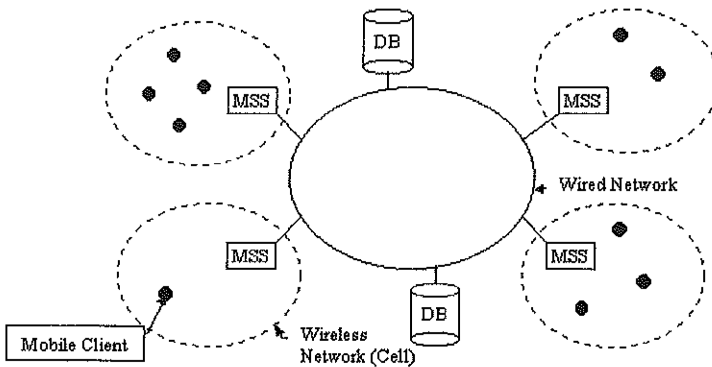
- **Query Languages and Processing:** Query translation may be required since query languages used to access the local DBMS may be different. Query processing and optimization is also difficult due to difference in data structures and processing power at each local DBMS. Requirements like conversion of data into standard format and processing queries at nodes with more processing power are some of the issues that merit consideration. Not only do these factors increase the overhead of query processing, but they can also create additional problems like communication bottlenecks and hot spots at servers, especially when available information regarding the local DBMS is inadequate at the global level. Fragmentation of data and incomplete local information can make developing an accurate cost model difficult, increasing the complexity of query optimization [55].

- **Transaction Processing:** In a MDBS, it is possible that the local DBMS may use different concurrency control and recovery schemes. It could happen that one database follows the two-phase locking protocol while others

use a timestamp-based scheme to serialize accesses. Furthermore, to maintain atomicity and durability of global transactions, the local database should support some atomic commitment protocol. A local DBMS joining a multidatabase environment may not have such a facility. The problem becomes even more severe if the local DBMS does not want to divulge or does not have any information regarding its local concurrency control and recovery schemes. This makes the task even more difficult for maintaining global consistency in a MDBS [9]. In a MDBS where updates are frequent, loss of correctness and inconsistency of data is often unacceptable. The database technology is built around the notion that data will be stored reliably and will be available to multiple users. Thus, maintaining the ACID properties of transactions is of vital importance.

4.2.4 MDAS Characteristics

A mobile data access system (MDAS) is a multidatabase system that is capable of accessing a large amount of data over a wireless medium. The system is realized by superimposing a wireless mobile computing environment over a multidatabase system [38]. The general architecture of the MDAS can be represented as shown in Figure 4.2.



MSS: Mobile Support Station
DB: Database

Figure 4.2. Mobile Computing Environment.

Mobile computing environment

The mobile computing environment is composed of two entities: a collection of mobile hosts (MH) and a fixed networking system [15,24,38]. The fixed networking system consists of a collection of fixed hosts connected through a wired network. Certain fixed hosts, called base stations or Mobile Support Stations (MSS) are

equipped with wireless communication capability. Each MSS can communicate with MHs that are within its coverage area (called a cell). A cell could either be a cellular connection, satellite connection, or a wireless local area network. A MH can communicate with a MSS if it is located within the cell governed by the MSS. MHs can move within a cell or between cells, effectively disconnecting from one MSS and connecting to another. At any point in time a MH can be connected to only one MSS. MHs are portable computers that vary in size, processing power, memory, etc.

Three essential properties pose difficulties in the design of applications for the mobile computing environment: wireless communication, mobility, and portability [24]:

- **Wireless Communication:** mobile computers rely heavily on wireless network access for communication. Lower bandwidths, higher error rates, and more frequent spurious disconnections often characterize wireless communication. These factors can in turn lead to an increase in communication latency arising from retransmission, retransmission time-out delays, error control protocol processing, and short disconnections. Mobility can also cause wireless connections to be lost or degraded. A mobile user may travel beyond the coverage area or may enter an area of high interference. Thus, wireless communication leads to challenges in the areas of:

  1. **Disconnection:** Wireless networks are inherently more prone to disconnection. Since computer applications that rely heavily on the network may cease to function during network failures, proper management of disconnection is of vital importance in mobile computing. Autonomy is a desirable property that allows the mobile client to deal with disconnection. The more autonomous a mobile computer is, the better it can tolerate network disconnection. Autonomy allows the mobile unit to run applications locally. Thus, in environments with frequent disconnections, it might be better for a mobile device to operate as a stand-alone device. In order to manage disconnection, a number of techniques such as caching, asynchronous operation, and other software techniques may be applied. Maintaining cache consistency is difficult however, since disconnection and mobility severely inhibit cache consistency. Cache consistency techniques employed in traditional architectures designed for fixed hosts may not be suitable for the mobile computing environment. Asynchronous operation can be used to mask round-trip latency and short disconnections. Software techniques such as prefetching and delayed writeback can also be used to minimize communication, thus allowing an application to proceed during disconnection by decoupling the communication time from the computation time of a program [2]. Delayed write back takes advantage of the fact that data to be written may undergo further modification. Operation queuing can also help; operations that cannot be carried out while disconnected can be queued and done when reconnection occurs.

2. **Limited Bandwidth:** Wireless networks deliver lower bandwidth than wired networks. Cutting-edge products for portable wireless communication achieve only 1 megabit per second for infrared communication, 2 Mbps for radio communication, and 9 - 14 kbps for cellular telephony. On the other hand, Ethernet provides 10Mbps, fast Ethernet and FDDI, 100 Mbps, and ATM (Asynchronous Transfer Mode) 155 Mbps [24]. Available bandwidth is often divided among users sharing a cell. Thus, bandwidth utilization is of vital importance. Software techniques such as compression, filtering, and buffering before data transmission can be used to cope with low bandwidth. Other software techniques such as prefetching and delayed-write back that are used to cope with disconnection can also help to cope with low bandwidth. A large dynamically changing number of mobile clients are a characteristic of a mobile computing environment. Thus, bandwidth contention is a problem. Caching can help to reduce bandwidth contention, which also helps to support a disconnected operation.

3. **High Bandwidth Variability:** bandwidth may vary many orders of magnitude depending on whether a mobile client is plugged in or communicates via wireless means. Bandwidth variability is treated by traditional existing systems as exceptions or failures [2]. However, this is the normal mode of operation for mobile computing. Applications must therefore have the ability to adapt to the available bandwidth and should be designed to run on full bandwidth or minimum bandwidth.

- **Mobility:** The ability to change location while retaining network connection is the key motivation for mobile computing. As mobile computers move, they encounter heterogeneous networks with different features. A mobile computer may need to switch interfaces and protocols; for example a mobile computer may need to switch from a cellular mode of operation to a satellite mode as the computer moves from urban to rural areas or from infrared mode to radio mode as it moves from outdoors to indoors. Traditional computers do not move, therefore, certain data that are considered to be static for stationary computing becomes dynamic for mobile computing. For example, a stationary computer can be configured to print from a certain printer attached to a particular print server, but a mobile computer needs a mechanism to determine which print server to use. A mobile computer's network address changes dynamically; its current location affects configuration parameters as well as answers to user queries. If mobile computers must serve as guides, location-sensitive information may need to be accessed. Thus, mobile computers need to be aware of their surroundings and have the ability to find location dependent information automatically and intelligently while maintaining system privacy [2]. Mobility can also lead to increased network latency and increased risk of disconnection. Cells may be serviced by different network providers and may employ different protocols. The physical distance may not reflect the true

network distance and therefore a small movement may result in a much longer path if a cell or network boundary is crossed. Transferring service connection to the nearest server is desirable but this may not be possible if load balancing is a key priority. Security considerations exist because a wireless connection is easily compromised. Appropriate security measures must be taken to prevent unauthorized disclosure of information. Encryption is necessary to ensure secure wireless communication, data stored on disks and removable memory cards should also be encrypted. The amount of data stored locally should be minimal; backup copies must be propagated to stationary servers as soon as possible as is done in replicated systems.

- **Portability:** Designers of desktops take a liberal approach to space, power, cabling, and heat dissipation in stationary computers that are not to be carried about. However, designers of mobile computers face far more stringent constraints. Mobile computers are meant to be small, light, durable, operational under wide environmental conditions, and require minimal power usage for long battery life. Concessions have to be made in each of the areas to enhance functionality [24]. Some of the design pressures that result from portability constraints include:

  1. **Low Power:** Batteries are the largest single source of weight in portable computers. Reducing battery weight is important, however too small a battery can undermine the value of portability leading to: i) frequent recharging, ii) the need to carry spare batteries, or iii) make less use of the mobile computers. Minimizing power consumption can improve portability by reducing battery weight and lengthening the life of the battery charge. Chips can be designed to operate at lower voltages. Individual components can be powered down when they become idle. Applications should be designed to require less communication and computation. Preference should be given to listening rather than transmitting since receptions consumes a fraction of the power it takes to transmit.

  2. **Limited User Interface:** Display and keyboard sizes are usually limited in mobile computers as a consequence of size constraints. The amount of information that may be displayed at a time is limited as a result. Present windowing techniques may prove inadequate for mobile devices. The size constraint has also resulted in designers abandoning buttons in favor of analog input devices for communicating user commands. For instance, pens are now the standard input devices for PDAs because of their ease of use while mobile, their versatility, and their ability to supplant the keyboard.

  3. **Limited Storage capacity:** Physical size and power requirements effectively limit storage space on portable computers. Disk drives, which are an asset in stationary computers, are a liability in mobile computers because they consume more power than memory chips. This restricts

Table 1. Characteristics of Mobile Environment and their effect on Database.

Mobile Characteristics		Resulting Issues
Wireless Connection ♠	♠♣	Disconnection
		Communication Channel
	♠	—High Cost
	♠	—Network Measurement
	♠	—Low Data Rate
Mobility ♣	♣	Motion Management
	♣	Location-Dependent Data
		Heterogeneous Networks
	♣	—Interfacing
	♣	—Data-Rate Variability
		Security
	♠	—Eavesdropping
	♣	—Privacy
Portability ♢	♢	—Vandalism
	♢	Limited Resources
	♢	Limited Energy Sources
	♢	User Interface

the amount of data that can be stored on mobile devices. Most PDA products on the market do not have disk drives. Flash EPROM, a dense, non-volatile solid state technology with a read latency close to that of a DRAM, and a write latency close to that of a disk that can withstand a limited number of writes over its lifetime is commonly employed. Solutions include compressing files systems, accessing remote storage over the network, shared code libraries, and compressing virtual memory [2].

Table 1 summarizes these issues and their effect on traditional issues of concern in a database environment.

4.2.5 MDAS Issues

The MDAS system is a multidatabase system that has been augmented to provide support for wireless access to shared data. Issues that affect multidatabases are therefore applicable to the MDAS. Multidatabase issues have received a lot of attention in the literature (see Section 4.2.3). Mobile computing raises additional issues over and above those outlined in the design of an MDAS. These issues are a consequence of the properties and inherent limitations of the mobile computing environment. In this section, we examine the effects of these properties on the issues of query processing, and optimization and transaction processing.

- **Query Processing and Optimization:** The reliance of the mobile client on battery power, the limited wireless bandwidth, frequent disconnection, and the mobility of the mobile client have an effect on how queries are processed. Query processing considerations need to take into account bandwidth considerations and communication costs. The existing query processing algorithms have focused mainly on resource costs. The fact that local area networks have become commonplace and the resultant lessening in importance of communication costs in this environment has led to this focus. Bandwidth limitation will motivate changes to query processing and optimization algorithms. The financial cost of wireless communication may lead to the design of query processing and optimization algorithms that focus on reducing the financial cost of transactions and consideration for query processing strategies for long-lived transactions that do not rely on frequent short communications, but longer communications instead. Query optimization algorithms may also be designed to select plans based on their energy consumption to limit the effects of database operations on the limited battery power. Approximate answers will be more acceptable in mobile databases than in traditional databases due to the frequent disconnection and the long latency time of transaction execution [2]. The issue of location dependent queries was discussed in Section 4.2.4.

- **Transaction Processing:** Since disconnection is a common mode of operation in mobile computing, transaction processing must provide support for disconnected operation. Temporary disconnection should be tolerated with a minimum disruption of transaction processing, and suspending of transactions on either stationary or mobile hosts. In order for users to work effectively during periods of disconnection, mobile computers will require a substantial degree of autonomy [2,38,57]. Local autonomy is required to allow transactions to be processed and committed on the mobile client. Effects of mobile transactions committed during a disconnection would be incorporated into the database while guaranteeing data and transaction correctness upon reconnection [57]. Atomic transactions are the normal mode of access to shared data in traditional databases. Mobile transactions that access shared data cannot be structured using atomic transactions. Atomic transactions execute in isolation and are prevented from splitting their computations and sharing their state and partial results. However, mobile computations need to be organized as a set of transactions, some of which execute on mobile hosts and others that execute on the mobile support hosts. The transaction model will need to include aspects of long transaction models and Sagas. Mobile transactions are expected to be lengthy due to the mobility of the data consumers and/or data producers and their interactive nature. Atomic transactions cannot satisfy the ability to handle partial failures and provide different recovery strategies, minimizing the effects of failure [2,14,61].

- **Transaction Failure and Recovery:** Disconnection and bandwidth limitations, and the mobile user dropping the mobile unit are some of the possible sources of failure in mobile environments. In a mobile unit, it is often the case that an impending disconnection and a drop in available bandwidth is predictable. Special action can be taken on behalf of active transactions at the time a disconnection is predicted. For example, transaction processes may be migrated to a stationary computer, particularly if no further user interaction is required. Remote data may be downloaded in advance of the predicted disconnection in support of interactive transactions that should continue to execute locally on the mobile machine after disconnection. Log records needed for recovery may be transferred from the mobile host to a stationary host; this is very important since stable storage is very vulnerable to failure due to the user dropping the machine [2].

4.3 Concurrency Control and Recovery

We begin by reviewing the meaning of a transaction and its role in concurrency control and recovery. It should be noted that this paper is not intended to address recovery issues in detail. A transaction is essentially a program that manipulates resources in a shared database or files. A transaction T_i consists of read r(x), write w(x) operations and either terminates by a commit operation c_i, making the effects of the transaction permanent, or by an abort operation a_i, erasing the effects of the transaction. A classical example of a transaction is the transfer of funds in a bank; e.g., a transaction in a bank may involve withdrawing of funds from a savings account and the subsequent depositing of the funds to a checking account.

In a multi-user environment, more than one transaction can access shared data simultaneously. In such an environment, synchronization is required to prevent undesired interference that can cause data inconsistencies. A simple example of an inconsistency due to interference is lost update. Assume transactions T_1 and T_2 read data item x, followed by writes to x by T_1, then T_2, without synchronization, the update to x by T_1 is lost. Another problem of interference is inconsistent retrieval. This occurs when a transaction reads one data item before another transaction updates it and reads some other data item after the same transaction has updated it. This scenario occurs when only some updates are visible, causing inconsistent retrievals or dirty reads [5].

The simplest solution to the interference problem is not to allow transactions to execute simultaneously. But, this in turn implies low throughput and poor utilization of resources, especially in the instance when transactions rarely access shared data simultaneously. Alternatively, one could allow concurrent execution of transactions and have algorithms that will synchronize accesses to shared data such that the final result is equivalent to some serial execution order of the transactions, i.e., serializability [6]. Serializability is widely used as a correctness criterion to ensure concurrency control since it is relatively simple to reason about serial executions compared to concurrent executions. In general, when referring to serializability, we

are concerned with a special case called conflict serializability. Conflict serializability means that conflicting operations of transactions are ordered in a serial fashion. Two operations are said to conflict if both of them access the same data item and one of the operations is a write operation. This in turn can give rise to direct or indirect conflicts between transactions.

- **Direct Conflict:** Two transactions T_i, T_j are said to be in direct conflict with each other if one or more of their operations conflict, denoted by, $T_i \longrightarrow T_j$.

- **Indirect Conflict:** Two transactions T_i, T_j are said to conflict indirectly if there exists transactions T_1, T_2, ... T_n such that $T_i \longrightarrow T_1 \longrightarrow T_2 ... \longrightarrow T_n \longrightarrow T_j$. If n = 0, then this reduces to a direct conflict. This type of conflict will be of particular importance to us later, when the serializability issues in multidatabases are discussed.

The most common method used to maintain serializability in a centralized database is the two-phase locking protocol (2PL) [6]. Locking is a pessimistic technique, since it assumes that transactions will interfere with each other and hence takes measures to synchronize accesses. Alternate schemes such as timestamp ordering, serialization graph testing [6], and optimistic concurrency control schemes performing commit time validation [36], have also been addressed in the literature. Table 2 summarizes the various concurrency control schemes.

Within the scope of transaction management, the recoverability of a database after failure should also be discussed. The atomicity property for a transaction dictates that either all or none of the transaction effect should be made permanent. In general, a transaction aborts if:

- The database is functional and it detects a bad input that can violate database consistency requirements,

- The transaction runs into a problem detected by the system such as deadlock or time-out,

- A system crash occurs causing any active transaction to be rolled back during recovery.

The basic requirement for recoverable execution is that a transaction can commit only after all previously active transactions that modified the values read by this transaction are guaranteed to commit. Recovery and atomicity issues have been dealt with by maintaining a log of the active and committed transactions that are used to undo effects of uncommitted transactions and redo the effects of committed ones [6].

Finally the problem of deadlock exists when resource conflicts occur. This is especially true when some sort of locking scheme is used for concurrency control. Deadlock usually occurs when a cyclic wait for a resource occurs among transactions. Deadlocks are usually dealt with by using time-outs to abort transactions [31]. An

Table 2. Concurrency Control Schemes.

Concurrency Control Scheme	**Description**	**Advantages and Disadvantage**
Two Phase Locking [5]	Two phases • Growing Phase: Acquire locks. • Shrinking Phase: Release locks.	• Pessimistic • Blocks Transactions - deadlocks can occur. • Most widely used in DBMSs.
Time Stamp Ordering [9]	Serialization is enforced using timestamps.	• May involve more restarts if serialization order is assumed a priori. • Memory requirement usually greater than locking methods.
Serialization Graph Testing [5]	Transactions are serialized by maintaining an execution history graph, and ensuring that this serialization graph is acyclic.	• Large memory overhead to maintain read-write sets of transactions used to detect conflicts.i
Optimistic Concurrency Control [30]	At commit time, transactions are validated to ensure serializability. Data conflicts are resolved by aborting transactions.	• Provides good performance for high data contentions systems, hardware resources are available. • May not be suitable for long transactions since it depends on transaction restart.

optimistic way of dealing with deadlocks is to break a deadlock when it occurs. In this case, a directed graph of transactions waiting for a particular resource has to be maintained. This graph is commonly called waits-for-graph (WFG) [6]. The WFG contains an edge $T_i \longrightarrow T_j$, if and only if T_i is waiting for T_j to release some lock. If a cycle is detected, some active transactions involved in the deadlock are aborted so that the deadlock can be broken. Having discussed the principles of serializability, atomicity/recoverability, and deadlock, we are now in a position to look at how these issues are translated into a multidatabase environment. The requirements in a distributed environment for global serializability, atomicity, and deadlock detection will thus become apparent.

4.3.1 Multidatabase Transaction Processing: Basic Definitions

Unlike centralized databases, in a distributed database system, there are two types of transactions: local and global. Local transactions are ones that are submitted at a local DBMS and executed locally, whereas global transactions are those that are submitted through the global interface and can potentially require execution at multiple local sites. The distinction is more relevant in a multidatabase system than a tightly coupled distributed database system. In a MDBS, local and global transactions are executed independently, whereas in a tightly coupled system the global manager has control over both local and global transactions; in a tightly coupled system there is no logical distinction between the two types of transactions.

In a multidatabase system, local transactions and global transactions generate three types of histories: local history, global subtransaction history, and global history [4].

- **Local History:** The local history (*LH*) is the history (*H*) at a particular local site consisting of local transactions and global subtransactions executing at that local site. Formally, a local history is a partial order with the ordering relation $<_H$ over transactions $T_1, T_2, ...T_n$ (including both local and global subtransactions) where:

  1. $LH = \cup_{i=1}^{n} T_i$
  2. $<_{LH} \supseteq \cup_{i=1}^{n} <_i$ i.e., execution order is maintained, and
  3. For any two conflicting operations p, q $\in$ *LH* either p $<_{LH}$ q or q $<_{LH}$ p, i.e., conflicting operations are executed serially.

- **Global Subtransaction History:** Global subtransaction history (GSH) at a local site is a subset of the local history at that site consisting of only global transaction (G) operations occurring at that site. Formally, global subtransaction history at site k is a partial order where:

  1. $GSH_k = \cup_{i=1}^{n} T_i$ where $T_i \in G_i$ i.e., T_i is the subtransaction of G_i at site k.
  2. $GSH_k \subseteq LH_k$

3. $<_{GSH_k} \subseteq <_{LH_k}$ i.e., GSH retains the order as specified in the local history.

- **Global History:** A global history is the union of all global subtransaction histories, with the order as reflected by the global subtransaction histories, i.e., the projection of the local histories over the global subtransactions.

Having defined the basic terms in multidatabase transaction processing, we explore the difficulty in maintaining serializability in a multidatabase system.

4.3.2 Global Serialisability in Multidatabases

The problem of maintaining serializability in a multidatabase is complicated by the presence of local transactions that are invisible at the global level. These invisible local transactions can cause indirect conflicts between global transactions that do not conflict based on global information. Furthermore, ensuring that transactions are serializable at each local site does not ensure global serializability.

To see the issue in more detail, consider a multidatabase made up of two local sites $LDBS_1$ containing data items a and b, and $LDBS_2$ containing data items c and d. Assume two global transactions G_1 and G_2

$$G_1 : r_{G_1}(a)w_{G_1}(c)$$
$$G_2 : w_{G_2}(a)r_{G_2}(d)$$

and two local transactions run at the local databases.

$$L_1 : r_{L_1}(a)r_{L_1}(b)$$
$$L_2 : r_{L_2}(c)w_{L_2}(d)$$

It is assumed that the local transaction manager at each local site guarantees serializability of transactions occurring at that site. The local histories are:

$$LDBS_1 : r_{G_1}(a)r_{L_1}(a)r_{L_1}(b)w_{G_2}(a)$$
$$LDBS_2 : r_{L_2}(c)w_{G_1}(c)r_{G_2}(d)w_{L_2}(d)$$

At $LDBS_1$, subtransactions of G_1 and G_2 are in direct conflict with each other; as a result, G_1 is serialized before G_2 giving the serialization graph shown in Figure 4.3. At $LDBS_2$, local transaction L_2 conflicts directly with subtransactions of G_1 and G_2 giving rise to an indirect conflict between them. The serialization graph is shown in Figure 4.4.

The projections of local histories onto the global transactions results in the order $G_1 \longrightarrow G_2$ at $LDBS_1$ and the order $G_2 \longrightarrow G_1$ at $LDBS_2$. Thus, the global history contains a cycle and is not serializable. This emphasizes the need to order multidatabase transactions not only in local DBMS where they directly conflict but also at other sites where they apparently do not conflict, but where hidden conflicts may arise due to the autonomy of the local sites [9].

Formally, the necessary and sufficient conditions for global history to be serializable is:

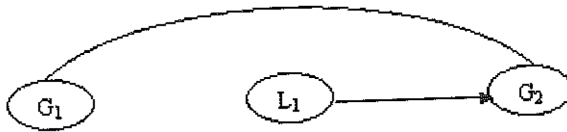


Figure 4.3. Serialization Graph at LDBS1.

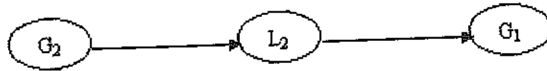


Figure 4.4. Serialization Graph at LDBS2.

1. Every local history LH in which the global subtransactions take place is serializable.
2. There exists a total order O on global transactions, such that at each local site the serialization order of global subtransactions is consistent with O. For example, if O for G_1 and G_2 is $G_1 \longrightarrow G_2$ then at every local site the subtransactions will have the same order.

In our running example, both local histories are serializable, but a total order on global transactions at local sites does not exist. As a result, the global schedule is non-serializable.

4.3.3 Multidatabase Atomicity/Recoverability

In a distributed environment, a global transaction is said to be committed if and only if it is committed at all the sites that its subtransactions execute. Traditionally, in a distributed database, a protocol such as the two-phase commit [12] is used to ensure atomic commitment. Briefly, the two-phase commit involves a global coordinator who sends a prepare-to-commit message to all the DBMSs participating in that transaction. Each participant in turn responds with a READY if it is ready to commit or ABORT if it is not. This constitutes the first phase. The coordinator in the second phase sends a COMMIT or an ABORT as the global decision, using the responses obtained in the first phase.

Multidatabase systems do not enjoy the luxury of inter DBMS communication and synchronization due to the autonomy of the local sites. Execution autonomy of the local DBMS implies that local DBMSs could implement a single-phase commit at their respective sites. The prepared to commit state or READY state is not explicitly available nor can it be trivially provided without modifying the underlying DBMS, a violation of local autonomy that may not be possible or desirable.

The consequence of dealing with the single-phase commit is illustrated by the following example. Consider a MDBS consisting of two LDBSs: $LDBS_1$ containing data item a, and $LDBS_2$ containing data item b. Let a global transaction G be initiated consisting of:

$$G{:}r(a)r(b)w(b)$$

Suppose the subtransactions of G at $LDBS_1$ and $LDBS_2$ are completed, after which a decision is made to globally commit G by the global transaction manager (GTM). Assume that the commit is globally issued, but before $LDBS_1$ can actually commit, a failure occurs at $LDBS_1$, whereas $LDBS_2$ successfully commits the subtransaction of G. Thus, the subtransaction of G is incomplete at $LDBS_1$ and hence during the recovery phase, the local DBMS will undo the effects of the active subtransaction of G at its site. We have a global state where one subtransaction of G is aborted while another is committed. This inconsistency is a direct consequence of the communication and execution autonomies of the underlying DBMSs. The committed transaction cannot be rolled back since its effects have been made permanent at $LDBS_2$. As an alternative, the global transaction manager (GTM) should redo the effects of subtransaction G at $LDBS_1$. While this looks straightforward, trouble may occur. Consider the scenario where, after recovery of $LDBS_1$, a local transaction $T_1(r_{T_1}(a)w_{T_1}(a))$ is allowed to execute before the subtransaction of G can be resubmitted.

In a MDBS system, this transaction is not visible to the GTM. The GTM, unaware of the local transaction that took place, will redo the subtransaction G in order to complete the missing write of G, $w_G(a)$, This in turn can lead to the following local history:

$$LH_1 : r_{T_1}(a)w_{T_1}(a)commit_{T_1}w_G(a);$$

had the DBMS failure not occurred, the history would have been:

$$LH_1 : r_G(a)w_G(a)commit_G r_{T_1}(a)w_{T_1}(a)commit_{T_1}.$$

Thus the net effect renders the MDBS schedule non-serializable, since the effect of the local write on a is not considered by the multidatabase recovery transaction. This occurs primarily because the local database does not know that on recovery, there are some global transactions that are pending. A local DBMS treats all transactions in the same manner and hence, on recovery, it just undoes incomplete transactions. If the local DBMS followed a two phase commit protocol, it would have realized that the subtransaction of G at its site was in a prepared state and would get the global decision (in this case commit) and try to redo G, disallowing local transactions from modifying data items accessed by G.

It has been shown that in general, it is impossible to perform atomic commitment without violating local autonomy [42]. Furthermore, atomic commitment (even under assumptions such as strict 2PL protocol at the local DBMS sites) is impossible

if the component DBMS are autonomous and there exists cyclical functional (commit, abort) dependencies among the subtransactions. Dependencies specify the effect of transactions on each other. In a multidatabase environment, there can be dependencies between the subtransactions of the same global transaction. Commit and abort dependencies are the two types of dependencies that may exist between subtransactions that can make atomic commitment difficult in a MDBS.

The dependencies that can occur between subtransactions of a global transaction are [13]:

- **Commit Dependency:** Subtransaction GST_1 is said to have commit dependency on GST_2 if GST_1 cannot commit until GST_2 either commits or aborts.
- **Abort Dependency:** Subtransaction GST_1 is said to have an abort dependency with subtransaction GST_2 if aborting GST_2 forces GST_1 to abort.

Thus, it is clear that atomicity is a problem in a MDBS environment, especially if there exists a dependency cycle among transactions and the local DBMS does not support some form of atomic commitment protocol.

4.3.4 Multidatabase Deadlock

If the component databases in a multidatabase use a blocking protocol like two phase locking to ensure global serializability, then global deadlock can occur. The problem is even more severe since the occurrence of a deadlock cannot be detected easily due to the lack of communication between the local and global transactions. The situation can be illustrated in the following example:

Consider a multidatabase composed of two local DBMSs: $LDBS_1$ with data items a and b and $LDBS_2$ with data items c and d. Assume that the local DBMSs use two phase locking to maintain serializability at their respective sites.

Let there be two global transactions

$$G_1 : w_{G_1}(a) r_{G_1}(c)$$
$$G_2 : w_{G_2}(d) r_{G_2}(d)$$

and two local transactions

$$L_1 : w_{L_1}(b) r_{L_1}(a) \text{ at } LDBS_1$$
$$L_2 : w_{L_2}(c) r_{L_2}(d) \text{ at } LDBS_2$$

Consider the sequence of events

- G_1 submits $w_{G_1}(a)$ to $LDBS_1$ and acquires lock on data item a.
- G_2 submits $w_{G_2}(d)$ to $LDBS_2$ and locks d.
- Local transaction L_1, submits $w_{L_1}(b)$ and acquires lock on b, however, for operation $r_{L_1}(a)$ it has to wait for G_1 to release the lock.

- Local transaction L_2 submits $w_{L_2}(c)$ and acquires lock on c, but for $r_{L_2}(d)$ it has to wait until G_2 releases d.
- Now if G_1 and G_2 submit $r_{G_1}(c)$ and $r_{G_2}(b)$, respectively, they will not be able to get a read locks on c and b, respectively.

What we have here is a cyclic wait, causing deadlock. To detect deadlock, we need to maintain a wait-for-graph involving local and global transactions but due to the autonomy requirements, the local DBMS may not divulge information regarding local transactions.

4.3.5 MDAS Concurrency Control Issues

The MDAS is based on a multidatabase, therefore, the basic concurrency issues that affect the multidatabase, which have been outlined in the preceding subsections (see Sections 4.3.3 and 4.3.4), are relevant and important, and equally apply. In a multidatabase, there are two types of transactions (global transactions and local transaction). The MDAS introduces an additional type of transaction - mobile transactions. A mobile transaction can be thought of as a global transaction. However, there are compelling reasons why mobile transactions should be viewed as a separate transaction type. A mobile transaction differs from other transaction types in the following ways:

- Mobile transactions might have to split their computations into sets of operations, some of which operate on a mobile host while others on a stationary host. Frequent disconnection and mobility result in mobile transactions sharing their states and partial results violating the principle of atomicity and isolation, which traditional transactions adhere to [40,44].
- Mobile transactions require computations and communications to be supported by stationary hosts [17,40,44]. Transaction execution may have to be migrated to a stationary host if disconnection is predicted in order to prevent the transaction from being aborted. The stationary host behaves like a proxy and executes the transaction on behalf of the disconnected mobile client. The mobile client may either fully delegate authority to the proxy to commit or abort the transaction as it sees fit or may partially delegate authority, in which case the final decision to commit or abort the transaction would be made by the mobile client upon reconnection.
- The states of transactions, the states of accessed objects, and location information move with the mobile host as it moves from cell to cell [17,40,44]. An interactive mobile transaction may be initiated at a site and completed at another site when the mobile unit moves. It is important that changes made prior to movement be visible at the new location. Otherwise, this could lead to out of date information being read by the mobile transaction at the new site, particularly if the earlier part of the transaction had written to the data item in question.

- Mobile transactions tend to be long-lived transactions due to the mobility of data consumers and/or producers and due to frequent disconnections [17,40,44]. If locking schemes are being employed, this results in the increased likelihood of deadlocks and aborted transactions. If optimistic schemes are being employed, this results in an increased likelihood of conflict and transaction restarts. The effect is the inadvertent decline in system throughput and increased response time of transaction execution. Concurrency control must minimize blocking and aborted transaction execution.

Transaction processing models must address the limitations of mobile computing. Concurrency control in the MDAS must therefore strive to minimize aborts due to disconnection. Operations on shared data must ensure correctness of transactions executed on both stationary and mobile hosts. Blocking or restarting of transactions must be minimized to reduce the communication cost as well as increase concurrency. Local autonomy must be supported to allow transactions to be processed on the mobile host despite temporary disconnection.

We have seen the major issues related to transaction processing that exist in a multidatabase and MDAS environments. In the next section, we will look at how solutions to these issues have been addressed in the literature. The complexity and difficulty of the problems have forced researchers to make some assumptions, even compromising autonomy in some cases. The solutions try to extend approaches from the tightly coupled distributed database systems to fit a loosely coupled environment. The approaches have their roots in the fact that a tightly coupled system is a special case of a multidatabase system.

4.4 Solutions to Transaction Management in Multidatabases

Solutions to transaction management problems in multidatabases can be broadly divided into two categories - those that ensure concurrency control in a failure free environment and those taking failure into consideration. The first category reduces the problem to that of ensuring database consistency in the presence of concurrently executing transactions. For solutions under this category, an assumption has to be made to ensure atomicity - the existence of an atomic commitment protocol such as the two phase commit protocol. The second category deals with correctness in the presence of failures, i.e., issues relating to atomicity, recoverability, and durability of multidatabase transactions.

In the previous section, we looked into the problems that arise in a multidatabase environment when attempting to maintain global serializability in the absence of knowledge of the transaction management scheme used by the component databases. Various solutions have been proposed in the literature to deal with this form of the transaction management problem. Some of these solutions use global serializability with conflict serializability as the correctness criteria. In environments where serializability can be a too strong and restrictive criterion to maintain, the use of weaker or relaxed criteria has been proposed.

Before actually discussing the existing solutions for generating serializable schedules in a multidatabase environment, we will look at some of the possible classifications for solving the concurrency control problem. As has been discussed, one of the important goals of a multidatabase system is to maintain autonomy of the local databases. It is precisely this goal that manifests the problem of maintaining global serializability. One school of thought may be that it is unrealistic to have global transaction correctness with no knowledge of the underlying databases. With maintaining correctness of data being an important goal in any database systems, it may be reasonable to assume that local databases joining a multidatabase system may be willing to cooperate as long they are not affected adversely. As a result, when a local database joins a multidatabase system, to aid global concurrency control, the LDBS allows the global system to make certain assumptions or compromises to the local autonomy. Another approach is to exploit the characteristics of the multidatabase environment. This knowledge can be used to extract the transaction semantics that can be exploited for concurrency control. The use of transaction semantics can allow interleaving of transactions that may lead to non-serializable execution, with the correctness criteria being the semantic consistency of the underlying databases. Serializability is a syntactic correctness criterion, whereas semantic based correctness criteria exploit transaction characteristics and hence can also increase the degree of concurrency. Semantic based methods are of particular relevance when dealing with atomicity, since non-blocking commit protocols can be formulated. Atomicity requires one to maintain the semantic atomicity of the transactions. Thus, we have five categories of solutions that deal with the global serializability problem, as illustrated in Figure 4.5.

It should be noted that global concurrency methods have also been categorized broadly as bottom up and top down methods [19].

- **Bottom Up Approach:** in this approach, collecting local information from the LDBSs and validating serialization orders at the global level verify global serializability. As a result, the LDBSs independently determine their own serialization orders and the global scheduler detects and resolves incompatibilities between global transactions and local orders.

- **Top Down Approach:** In this approach, the global scheduler is allowed to determine a serialization order for global transactions before they are submitted to the local sites. It is the responsibility of the LDBSs to enforce this order at their respective local sites, by:

  - Controlling the submission of global subtransactions if the underlying local concurrency control scheme is known, or

  - Modifying the local schedulers.

  This is a violation of local autonomy: therefore, the top down approach can be applied only when the LDBSs can tolerate a certain degree of autonomy violation.

The top-down approach is generally pessimistic in the sense that global order is forced on the local databases. It should be noted that the top-down and bottom-up parameter used to classify the scheduling policy motivates generic approaches without emphasizing any correctness criteria or methods used in the MDBS formation. Hence, it is possible that the solutions in the initial classification (Figure 4.5) could use either bottom-up or top-down policy.

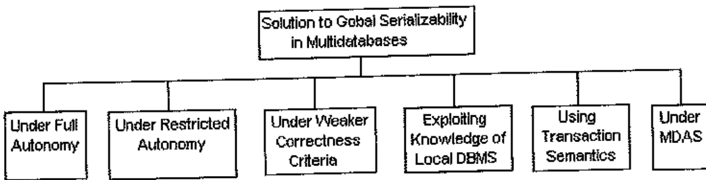


Figure 4.5. Solutions To Multidatabase Serializability.

4.4.1 Global Serializability under Complete Local Autonomy

Definition 1 Full Autonomy from the concurrency control point of view implies the global scheduler does not have:

- Any knowledge of local transactions occurring at the local databases. This in turn implies that the global scheduler does not have any knowledge of indirect conflicts that could possibly occur.
- Any knowledge of the concurrency control scheme used by the local databases.
- Any authority to enforce modifications on the local schedulers in order to facilitate global concurrency control.

Under circumstances of full autonomy, all we can expect from local databases is that they will maintain serializability of transactions occurring at their respective sites. It is the responsibility of the global transaction manager to ensure serializability of global transactions. Hence, full autonomy solutions fall under the bottom up approach of concurrency control.

We will look at the solutions under this category using the example employed to illustrate the global serializability problem. The following recaps the example given in the previous section. The problem occurred because the projection of global serialization order at one local site resulted in:

$G_1 \longrightarrow G_2$,

and at the second local site, the projection of serialization order resulted in:

$G_2 \longrightarrow G_1$,

indicating a cycle in the global serialization graph. If we look at the local transaction history at $LDBS_2$ we see: $LDBS_2 : r_{L_2}(c)w_{G_1}(c)r_{G_2}(d)w_{L_2}(d)$.

$$LDBS_1\text{: Data items a,b}$$
$$LDBS_2\text{: Data items c,d}$$
$$LH_1 : w_{G_1}(a)r_{G_2}(a)$$
$$SerializationGraph_{LDBS_1} : G_1 \longrightarrow G_2$$
$$LH_2 : w_{G_2}(c)r_{G_1}(c)$$
$$SerializationGraph_{LDBS_1} : G_2 \longrightarrow G_1$$

Figure 4.6. Global-Global Conflicts.

Although the transactions G_1 and G_2 were executed in the desired serialization order, $G_1 \longrightarrow G_2$, an indirect conflict caused by the local transaction L_2 reversed the serialization order at $LDBS_2$. The global transaction manager has information regarding the global transactions only. Therefore, the local transactions are like phantom transactions. Another problem that was not discussed in the previous section was the result of non-serializable schedules caused by direct conflicts between global transactions as illustrated in Figure 4.6.

The only solution to serializing transactions under full autonomy and in the presence of indirect conflicts is to assume that global transactions conflict at every local site they execute concurrently. It is easier to deal with non-serializable transactions caused only by direct conflicts between global transactions since the global transaction manager has full knowledge of all global operations. Solutions under full autonomy are given below and summarized in Table 3.

Site Graph Method: The site graph method, proposed in [8], is a pessimistic method towards global concurrency control in multidatabase systems. A site graph is an undirected graph, with nodes being the sites containing the data items accessed by the global transactions and edges corresponding to the sites spanned by the global transaction. Initially, there are no edges between nodes in the graph. When a global transaction issues a subtransaction, undirected edges are added between nodes of the LDBSs that participate in the execution of the global transaction. As long as the site graph is acyclic, serializability of global transactions is guaranteed. A cycle in the site graph indicates a possibility of a non-serializable schedule and hence, the global transaction has to be aborted.

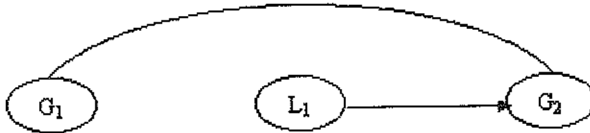
In our running example, when G_1 executes at local sites 1 and 2, an edge is inserted between nodes 1 and 2. Now, when G_2 submits its operation at sites 1 and 2, another edge is inserted between the two nodes, causing a cycle in the site graph. As a result, G_2 is aborted.

The site graph method is an example of a bottom up concurrency control scheme because the site graph is a global data structure used by the GTM. Note that the GTM is the only unit responsible for global concurrency control in this approach. The site graph method preserves the autonomy of the local databases but suffers from low concurrency, since a cycle in the site graph does not necessarily imply a non-serializable global schedule. The throughput is also reduced, since only one transaction can execute concurrently at the same sites. Another disadvantage is

due to the method that is used to safely remove an edge from the site graph. At first, it seems that once a transaction commits, the edges corresponding to that transaction can be removed from the site graph. But conflicts can occur even after a transaction commits, resulting in non-serializable executions, as illustrated in [6]. The knowledge about the local concurrency control mechanism, or weaker notions of correctness such as quasi-serializability [16], can be used to develop a safe policy for edge removal. The site graph method has been used for concurrency control in the Amoco Distributed Database System (ADDS) [10].

Forced Conflict Method: In the forced conflict method [30], the global scheduler generates globally serializable schedules by assuming that the local databases generate locally serializable schedules: serializability among global transactions is ensured by using a special data item, called a ticket, at every local DBMS where they execute. As we have seen, global transactions that appear globally serializable may generate globally non-serializable schedules due to indirect conflicts caused by local transactions. If all concurrent global transactions that span over the same sites are forced to conflict directly, then the problem due to indirect conflicts does not arise, since the induced conflicts serialize the global transactions. Each global subtransaction is modified to read and write the ticket data item at each local database they access. The ticket operation forces direct conflict between global subtransactions, irrespective of the other operations contained in the global subtransactions - the ticket acts as a (logical) timestamp and is stored as a regular data item in each LDBS(Figure 4.7).

$LH_1 : r_{G_1}(t_1)w_{G_1}(t_1)r_{G_1}(a)r_{L_1}(a)r_{L_1}(b)r_{G_2}(t_1)w_{G_2}(t_1)w_{G_2}(a)$



$LH_2 : r_{L_2}(c)r_{G_1}(t_2)w_{G_1}(t_2)w_{G_1}(c)r_{G_2}(t_2)w_{G_2}(t_2)r_{G_2}(d)w_{L_2}(d)$

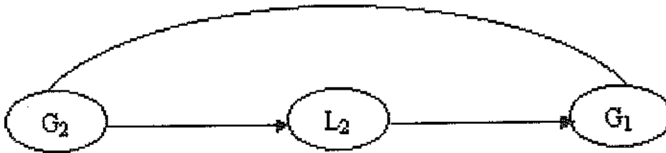


Figure 4.7. Example of using tickets.

In our running example (Figure 4.7), additional operations to modify the ticket are required in the local schedules, shown as follows: where t_1 and t_2 are the tickets for $LDBS_1$ and $LDBS_2$, respectively.

The history at $LDBS_2$ contains a cycle. Since the local schedules are serializable at their respective sites, the schedule at $LDBS_2$ is not allowed by the local

concurrency control mechanism. As a consequence, to maintain local serializability, one of the transactions will be aborted or blocked. Furthermore, ticket values read are used to determine the relative serialization order among global subtransactions. It could happen that at $LDBS_1$, $GST_1 \longrightarrow GST_2$ whereas at $LDBS_2$, $GST_2 \longrightarrow GST_1$. Both of these executions orders are valid at their respective sites. At $LDBS_1$, GST_1 executed before GST_2 and the ticket operations created the conflict $GST_1 \longrightarrow GST_2$ whereas, at $LDBS_2$, GST_2 is executed before GST_1 giving $GST_2 \longrightarrow GST_1$. But globally, the relative serialization orders as indicted by the ticket values read at the respective sites are not consistent and hence one of the global transactions is aborted.

The autonomy of the local DBMS is preserved, since no change is required in the local DBMS to support the ticket operation. Also, the local transactions are not affected by the ticket operations, since only subtransactions of global transactions have to read and manipulate the ticket values. The disadvantage of this method is that the ticket operation causes conflicts between global transactions even when they may not actually conflict. This can lead to numerous aborts, especially when optimistic scheduling is employed. Also, the ticket data item could become a hot spot [6] in the local database, if several transactions try to access the database simultaneously.

A rare problem (but not too severe) is related to the autonomy of the local databases if the local DBMSs do not support the creation of the ticket data object. This problem can easily be resolved by adding additional operations in the transactions, which will induce direct conflicts among the transactions. However, this solution could result in more aborts than the pure ticket method, since the newly added operations reference a data item accessible by both global and local transactions.

A forced conflict method is an important algorithm for multidatabase concurrency control since it demonstrates that global serialization can be achieved by using the fact that the LDBS concurrency control method will serialize transactions at their respective sites. However, strict serializability is very restrictive and may even be inappropriate under certain circumstances [47]. Therefore, we will look at solutions that maintain autonomy of the component databases under relaxed (serializability) correctness criteria.

4.4.2 Solutions using Weaker Notions of Consistency

Serializability can be too strong a criterion, especially in a multidatabase environment where the underlying databases are autonomous. The low degree of concurrency obtained could significantly increase transaction response time; this could also increase delays among transactions. As a consequence, weaker notions of consistency such as quasi serializability [16] or multidatabase serializability [4], and two-level serializability [41] have been proposed. Generalizations of serializability such as Epsilon serializability [49], to allow a bounded amount of inconsistency, would result in a higher degree of concurrency. Before proceeding further, we will

Table 3. Concurrency Control Under Full Autonomy.

Solution	Description
Site Graph Method [8]	• Maintains a site graph to serialize global transactions. • If more than one global transactions spans over the same sites, it allows only one of the transactions to execute.
Forced Conflict Method [30]	• Global transactions are made to directly conflict at all local sites. • Local databases maintain a special data item - ticket - that is used to serialize global transactions.

briefly explain the notion of quasi-serial histories, two-level serializable histories, and Epsilon serializability.

Two-Level Serializability (2LSR):

Definition 2: A schedule is two-level serializable if:

- The schedule restricted to each local site is serializable.
- The projection of the global schedule onto the set of global transactions is serializable, i.e., the schedule excluding the local transactions is serializable.

Two-level serializable schedules are a superset of conflict serializable schedules (2LSR $\supset$ CSR). It has been shown in [41] that 2LSR schedules preserve database consistency only for certain applications. The basic idea is to exploit the knowledge of the inter-site data dependencies to relax restrictions on the global transactions and find schedules that use 2LSR and maintain database consistency. Consistency is preserved by capturing inter-site integrity constraints and by partitioning the data into global and local data items. A data item is a global data item if there is an inter-site integrity constraint between it and a data item at a different site. It should be noted that partitioning the data imposes restrictions on the data items read and written by a local/global transactions.

Quasi-serializable schedules are a subset of 2LSR schedules. Similar to 2LSR, quasi- serializable schedules also rely on relaxing the serializability criterion.

Quasi Serializability:

Definition 3: A global history is quasi serial if all the local histories are conflict serializable and there exists a total order of all global transactions such that, for any two global transactions G_i and G_j, if G_i precedes G_j, in the global order then all of G_i's operations precedes all of G_j's operations in all local histories in which both of them appear.

A global history is quasi serializable if it is equivalent to a quasi-serial history. Quasi serializability preserves database consistency, if i) there are no value dependencies between data items stored at different databases, i.e., if X = Y + Z then X and {Y, Z} are not allowed to be at different databases, and ii) integrity and consistency constraints are defined locally for each local database. These conditions may be reasonable if we consider the manner in which a MDBS system is typically developed - a collection of independent and autonomous databases. In fact, in [28] it has been argued that having a global constraint could be a violation of local autonomy; by agreeing to enforce a global constraint, a local database is at the mercy of the global transaction manager or other nodes in the MDBS system for control over its own data items. Thus, in a MDBS system, it is highly unlikely that two databases that join the system will have any inter-site integrity constraints or value dependencies (such as referential integrity constraints). Multidatabase serializability (introduced in [4]) is equivalent to quasi serializability and hence is not described here.

Comparing the different correctness criteria (i.e., conflict serializability, quasi serializability and two-level serializability), quasi serializability is a subset of two-level serializability and is more restrictive, while conflict serializability is a subset of both two-level and quasi serializability and is the most restrictive. The relationship between the correctness criteria is illustrated in Figure 4.8.

Using our running example, it is worth discussing the use of quasi serializability as the correctness criterion. In the schedule at $LDBS_2$, if subtransactions of G_2 are submitted after subtransaction of G_1 completes, then the total order (G_1 preceding G_2) among the global transactions is maintained. Indirect conflict does not affect global database consistency due to the absence of value dependencies. Thus, if we were using the site graph method at the global level, as soon as G_1 completes, the edge corresponding to G_1 can be removed and G_2 can be submitted - the weaker notion of consistency enables early removal of edges from the site graph. Concurrency control using methods such as quasi serializability are advantageous if the local databases are truly independent of each other since, under such circumstances, no restrictions are imposed on the underlying local databases and the local autonomy is preserved.

4.4.3 Solutions Compromising Local Autonomy

Until now, we looked at solutions that try to preserve local autonomy either by using strict correctness criteria or by using weaker consistency notions. Weaker notions of consistency do need a few assumptions, but in general can be applied without imposing any design changes to the local database. The schemes that

compromise local autonomy imposes some modifications on local databases when joining a multidatabase system in order to aid global concurrency control.

One of the early efforts that compromised the design autonomy of local databases was given in [46]. This approach requires strict serializability as the correctness criteria for local as well as global concurrency control. The scheme uses a data structure, order element (O-element), to represent the serialization order of subtransactions in component databases. The order element is determined depending on the concurrency control method being used by the underlying local database. For example, if transactions are serialized by timestamps at a local site, then the timestamps can serve as the O-element. If we have the local serialization order to be $T_i \longrightarrow T_j$, then according to concurrency control mechanism O-element $(T_i) \longrightarrow$O-element(T_j), A data structure called an order vector is formed by concatenating order elements from the component databases. Having formed the order vector, it is possible to analyze the vector to maintain serializability. This method, implemented in the Harmony multidatabase project at Columbia University [45], is an example of a bottom up approach to concurrency control, since the global transaction manager is responsible for enforcing global concurrency control.

A top down approach to concurrency control that also modifies local DBMSs is described in [19]. A sub process that serves as an interface between the multidatabase and the underlying local DBMSs is used to serialize transactions. The order of global transactions is maintained by controlled submission of global subtransactions to the LDBS. The approach utilizes the local concurrency control schemes in order to enforce serializability.

Alternatively, a top-down approach to a concurrency control scheme can be employed by modifying the local schedulers. Modified local schedulers serialize both global and local transactions. The top down approach does not achieve maximum concurrency since the serialization order is predetermined at the global level. Because runtime ordering of global transactions is disallowed, only the local histories compatible with the predetermined global order are allowed.

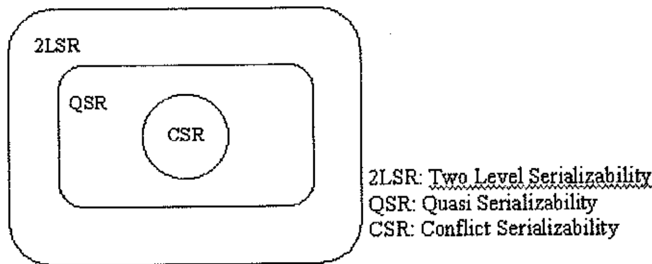


Figure 4.8. Relationship between Different Serializability Schemes.

4.4.4 Using Knowledge of Component Databases

To this point, we have discussed concurrency control schemes where no knowledge of local concurrency control mechanism was available. If the local database joining the MDBS can make its local concurrency control scheme known to the global manager, this information can be exploited for global concurrency control with few or no additional compromises to the local autonomy. In the remainder of this subsection, we look at how the local concurrency control scheme can be exploited to maintain global serializability. Timestamp ordering and strict two- phase locking protocols are used to motivate the discussion.

Timestamp Ordering: A timestamp ordering (TO) scheduler organizes conflicting operations according to their timestamps. A TO scheduler assigns a unique timestamp, ts(T_i), to each transaction T_i to serialize it, using the TO rule. By the TO rule, if T_i and T_j. have conflicting operations, then the operation of T_i is processed earlier than that of T_j if, and only if, ts(T_i) $<$ ts(T_j).

In our running example, a problem is caused by the schedule at $LDBS_2$ because of the indirect conflict caused by local transaction L_2. If timestamp ordering were to be followed at $LDBS_2$ then as transactions are submitted, they would be assigned timestamps. In our example L_2 was the first transaction at $LDBS_2$ and hence would be assigned the lowest timestamp, followed by G_1 and G_2, The problem caused by the last operation of L_2 would not exist at $LDBS_2$ since the last operation of L_1 would have been disallowed by the TO scheduler because conflicting operations of G_1 and G_2 with ts(G_2) $>$ ts(G_1)$>$ ts(L_2) already have been scheduled, making this operation too late [6].

Strict Two-Phase Locking: A scheduler is said to be using two-phase locking if a transaction can be divided into two phases - a growing phase during which the transaction obtains locks and the shrinking phase during which the locks are released. Strict two-phase locking is a variant of two phase locking in which a transaction holds onto its locks until it terminates - all the locks held by a transaction are released on termination (commit or abort).

In our running example, the problem was caused by the history at $LDBS_2$ where the schedule was $LH_2 : r_{L_2}(c)w_{G_1}(c)r_{G_2}(d)w_{L_2}(d)$.

With a strict two-phase locking scheduler at $LDBS_2$, L_2 will obtain a read lock on data item c when it submits the operation $r_{L_2}(c)$. Strict two-phase locking protocol ensures that L_2 will not release the read lock on c until the transaction terminates. Therefore, when G_1 submits w(c), it is unable to get write lock on data item c. To serialize global transactions, if the global scheduler assumes that an indirect conflict can always occur due to local transactions, it would then submit global subtransactions in a serial fashion. Since subtransactions of G_1 and G_2 both access $LDBS_1$ and $LDBS_2$, a conservative global scheduler would delay subtransactions of G_2 until G_1 terminates (commits or aborts). The fact that G_1 has completed and the knowledge that local sites use the strict-two phase locking protocol guarantees serializable transactions.

Functionality of local histories and the possibility of achieving global concurrency

control based on such functionality are summarized in Table 4. We have seen that the knowledge of the underlying local concurrency control scheme can aid in global concurrency control. A more detailed discussion of the use of local concurrency control schemes for global concurrency control can be found in [9].

4.4.5 Global Serializability Based on Transaction Semantics

Non-serializable transactions may be allowed if transaction semantics are taken into consideration. The idea is to exploit transaction semantics to specify compatibility between operations. Transactions are divided into a collection of disjoint classes such that transactions belonging to the same class are compatible. In other words, the transactions can interleave arbitrarily. Transactions that belong to different classes are incompatible; that is, they cannot interleave [22]. Consequently, the history that is produced due to execution of the transactions is equivalent to a correct schedule or has an equivalent semantic effect - semantically serializable. An example of this is the case where one has two deposits and the order in which amounts get deposited is immaterial as long as the effects of both transactions get reflected. Knowledge of the semantics of the operations and the manner in which they can commute can be specified in terms of forward and backward commutativity of transaction operations. The requirement for semantic concurrency control is that a compatibility table has to be provided by the local database administrator so that semantic conflict tests can be performed. This may be considered a violation of local autonomy, but it can be done as an abstraction above the existing concurrency control schemes. If the required information is unavailable, then no harm is done since concurrency control can still be performed using traditional schemes based on strict serializability.

4.4.6 Solutions under MDAS

MDAS Transaction Management deserves a classification of its own due to the peculiar nature of the environment. Though most of the solutions fall somewhat under the general classifications above (see Figure 4.5), however, the uniqueness of the MDAS warrants a separate classification of its own.

Multidatabase Transaction Processing Manager (MDSTPM): The model proposed in [50] augments the software component of the multidatabase system by adding a Global Interface Manager (GIM) that coordinates the submission of request/reply between the MDSTPM and the local database manager.

The approach used makes the mobile unit part of the multidatabase during its connection with the respective coordinator node, the MSS (see Figure 4.2). The MSS can schedule and coordinate the execution of the global transaction on behalf of the mobile unit upon submission of the global transaction. This is based on the following rational:

1. The user may disconnect from the network and perform some other task without having to wait for the global transaction to complete and,

2. The host computers are connected via reliable communication networks and are thus less prone to network failures.

A Message and Queuing Facility (MQF) approach is proposed to facilitate the implementation of the strategy. Traditionally, a Remote Procedure Calls (RPC) mechanism has often been used for interprocess communication in distributed computing environments. In the RPC paradigm, an application program requests services from another application executing in a remote node. The strategy is similar to that of a subroutine call in a program. The implementation would imply that events are occurring synchronously as the caller would have to wait for the control to be returned before continuing its process. This has the disadvantage that the mobile unit cannot disconnect until the results are returned back to it, otherwise, the transaction would be aborted. MQF allows asynchronous operation instead. Messages are handled asynchronously allowing the mobile unit to disconnect from the network to perform some other tasks leaving the coordinating node to manage the execution of the global transaction submitted on its behalf. In MQF, a mobile unit sends a request message, together with the information required for processing, to its pre-assigned coordinating node for processing.

The MQF strategy is deemed to be most appropriate because of the following advantages:

1. It is simple to manage the delivery and recovery of messages.
2. It is time independent since mobile units may be disconnected for an unbounded period of time while the global transaction submitted by these mobile units are being coordinated and executed by their respective coordinating nodes.
3. The ability of each workstation to query the status of its global transactions at its convenience.

This approach would fall under the classification of solutions under full autonomy since no knowledge of the underlying database management systems is required or assumed by the method.

Kangaroo Transaction Model: The kangaroo transaction (KT) model defines mobile transactions based upon global transactions in a multidatabase system and split transactions [17,48]. The mobile DBMS is viewed as an extension of a distributed system. It derives its name from the migrating "hopping" nature of mobile transactions. It requires a software component at the MSS that acts as the coordinator of global transaction for mobile computers that connect to the MSS, called a Data Access Agent (DAA). The goals of the model as outlined are:

1. Build on existing multidatabase systems and do not duplicate support provided by a source system.
2. Capture movement of mobile transaction as well as data access. Move transaction control as mobile unit moves.

Table 4. Relevance of Local Properties for Global Concurrency Control.

History	Description	Relevance to Global Serializability
Recoverability [6]	A transaction commits, only after the commitment of all transactions from which it read.	This property alone cannot be used to maintain global serializability.
Avoiding Cascading Aborts [6]	A transaction may read only those values that are written by committed transactions.	This property alone cannot be used to maintain global serializability.
Strict[6]	No data item may be read or overwritten until the transaction that previously wrote into it is terminated (committed or aborted).	This property alone cannot be used to maintain global serializability.
Rigorous [7]	No data item may be read or overwritten until the transaction that previously read or wrote that item is terminated (committed or aborted)	With proper scheduling, global serializability can be maintained.

3. Provide flexibility in terms of atomicity feature.

4. Support for long lived transactions.

The KT model is built on traditional transactions - a sequence of operations executed under the control of one DBMS. The view of global transactions is much broader than what is normally assumed. Two types of global transactions are considered: the limited view where a global transaction is composed of subtransactions, which can be viewed as local transactions to some existing LDBMS, and the broader view, where subtransactions may also be global transactions to another multidatabase system.

The DAA creates a mobile transaction upon receipt of a request from the mobile user, called a Kangaroo Transaction (KT), at the associated MSS. Each subtransaction represents the unit of execution at one MSS and is called a Joey Transaction (JT). A JT is part of a Kangaroo Transaction and it must be coordinated by a DAA at some base site. When the mobile unit migrates from one cell to another, the control of the KT changes to a new DAA at the new MSS. The new DAA at the new MSS subsequently creates a new JT as part of the handoff process. The

creation of the new JT is accomplished by a split operation. The old JT is committed independently of the new JT. The failure of a JT may cause the entire KT to be undone at any time; this is accomplished by compensating any previously completed JTs since the autonomy of the local databases must be assured. Two different processing modes for KTs are available: Compensating Mode and Split Mode. Under the compensating mode, the failure of any JT causes the current JT and any preceding or following JTs to be undone and previously committed JTs to be compensated for. Operating in this mode requires that the user provide information needed to create compensating transactions. The split mode is the default mode and under this mode, when a JT fails, no new global or local transactions are requested as part of the KT. In addition, any previously committed JTs will not be compensated for. Neither mode guarantees serializability of kangaroo transactions. Although the compensating mode ensures atomicity, isolation may be violated because locks are obtained and released at the local transaction level; however, JTs are serializable. This approach would thus fall under the classification of solutions that use transaction semantics.

V-locking Transaction Model: The V-locking algorithm was proposed in [38] and uses a global locking scheme in order to serialize the conflicting operations of global transactions. Global locking tables are used to lock data items involved in a global transaction in accordance to 2PL (Two Phase Locking) rules. In typical multidatabase systems, maintaining a global locking table would require communication of information from the local site to the global transaction manager regarding locked data items. This is impractical due to the delay and the amount of communication involved.

Under this method, the MDAS is a collection of summary schema nodes (SSM) [11] and local databases distributed among local sites. The MDAS software is distributed in a hierarchical structure similar to the hierarchical structure of the SSM. Transaction management is performed at the global level in a hierarchical, distributed manner. A global transaction is submitted at any node in the hierarchy. The transaction is resolved and mapped into subtransactions by the summary schema structure. The resolution of the transaction also includes the determination of the coordinating node within the structure of the SSM with the coordinating node being the lowest summary schema node that semantically contains the information space manipulated by the global transaction.

The MDAS coordinates the execution of global transactions without any control information from the local DBMS. The only information required by the algorithm is the type of concurrency control performed at the local sites. The semantic information contained in the summary schemas is used to maintain global locking tables. The locking tables can be used in an aggressive manner when the information is used only to detect potential deadlocks. A more conservative approach could be used as well, where the operations are actually delayed until a global lock request is granted. The global locking tables are used to create a global wait-for graph that is used to detect and resolve potential global deadlocks. The accuracy of the "waiting information" contained in the graph is dependent on the amount of communication

overhead that is required. The algorithm can dynamically adjust the frequency of communications (acknowledgment signals) between the GTM and local sites, based on network traffic and/or a threshold value.

The algorithm provides higher reliability but at the expense of lower throughput if an aggressive approach is used since it is based on the application of semantic contents rather than exact contents. Decrease in communication between the local and global systems comes at the expense of an increase in the number of potential false aborts but this must be weighed against the cost of communication.

To this point, we have discussed the issues of the correctness of a transaction when concurrent transactions occur in a failure free environment. The atomicity and recoverability issue is equally important. In fact, global serializability is affected by transaction aborts and failures. In the following discussion, we look at solutions to the global atomicity and recoverability problem in multidatabases.

4.4.7 Solutions to Global Atomicity and Recoverability

Maintaining atomicity of global transactions is difficult in a multidatabase environment due to the autonomy of the local databases. The problem arises due to the fact that the individual local databases may not follow an atomic commitment protocol such as the two-phase commit protocol. As a result, the local databases can unilaterally abort a global subtransaction. This independence is of particular concern in a situation where the global decision was to commit a global transaction, but before the decision reaches the local databases, a failure occurs. Since the local database does not follow an atomic commitment protocol, the active global subtransaction is aborted, in contradiction to the global decision. This could lead to a situation where few of the global subtransactions are committed and a few other subtransactions of the same global transaction have been aborted.

Traditional approaches used for recovery follow a redo or undo approach to maintain atomicity of transactions [6]. A redo approach to multidatabase recovery would be to resubmit the global subtransaction, as a corrective action, at the local site where the transaction was aborted. An undo approach to global recovery would be to abort the global transaction, which would involve rolling back the already committed global subtransactions. However, this is not possible due to the semantics of the commit operation [6].

Several approaches have been proposed in the literature to deal with the global recovery problem. Some of the solutions impose restrictions on the local DBMS, while others put restriction on the transactions. We look at the various solutions to deal with this problem in the remainder of this section. We begin with a method that attempts to maintain atomicity of the global transaction using an approach similar to the existing traditional two phase-commit protocols.

Simulated Two Phase Commit: Atomic commitment in a distributed DBMS is critical. Two-phase commit (2PC) is the most commonly used atomic commitment protocol in distributed databases. However, the 2PC scheme is suitable for a tightly integrated distributed database system where the underlying DBMSs are

willing to communicate with the GTM.

In a MDBS system, the local DBMSs generally follow a single-phase commit due to the inherent autonomy of the local databases; i.e., local databases do not support an explicit prepared state. Therefore, the challenge in a MDBS environment is to simulate a prepared-to-commit or READY state similar to that in the traditional 2PC protocol. After completing their operations at a local site, the global subtransactions should wait until a global decision to commit or abort the transaction arrives. The basic idea behind simulation of a prepared state is that at some point during execution, a transaction will test a condition to see if the GST should proceed or abort [3]. Thus to simulate a 2PC protocol, an operation is added to the global subtransaction that forces it to send a message to the GTM after completion of all its operations. After that, a global subtransaction waits for a reply from the GTM. This state can be viewed as the equivalent to the prepared-to-commit state. A simulated 2PC protocol can maintain atomicity as long as the data items accessed by the global subtransaction are not modified by some other transaction(s) before the global transaction has committed. To avoid interference, the underlying database should not allow any other transaction to access the data items required by the global transaction. In the prepared state, the local DBMS should deny accesses to the data items accessed by the global transaction in the prepared state until that transaction commits.

The simulated 2PC method maintains atomicity in the absence of failures. However, in the presence of failures, the problem created by the lack of knowledge of the global subtransactions at the local DBMSs still exists. Even if the global subtransaction were in a prepared state, the local DBMS unrolls it like any other active local transaction at the time of failure.

The recovery issue after failure can be dealt with by:

- Putting restrictions on the data in the local DBMS,
- Giving the multidatabase system control on recovery, or
- Using methods that lead to eventual consistency - retrying the transaction or using compensating semantic actions.

Partitioning Local Data: To preserve multidatabase consistency, one could divide the database into two mutually exclusive sets of data items: local and global [56,60]. The approach to partition data use is called denied local updates property. As a result of partitioning, local transactions do not update objects read or written by a global subtransaction. Similar to the simulated 2PC method, the denied updates property, along with the two-phase agent method (with the additional restriction that local histories must be strict), can be used to preserve multidatabase consistency [60]. To enable recovery, a log is maintained at the global level for global subtransactions that are in a prepared state (simulated). If a site failure occurs, this log is used to resubmit global transactions in the prepared state. As a result, consistency is maintained at the expense of violating the local autonomy and severe

limitations on the types of global transactions - existing data cannot be accessed if it is not in the global set.

Using MDBS Control on Recovery: One problem encountered during recovery in multidatabases is that the local transactions can start executing even before the global recovery actions have been completed, causing non-serializable global executions. Intuitively, the problem could be circumvented if the local transactions are not allowed to execute until global recovery actions have been completed. On completion of global recovery, a handshake between the global and local managers is exchanged to enable the local database for local/global transactions. This approach (giving MDBS control on recovery) has been followed in [3] and [29]. In addition to the local log at each local site, a global log containing information on global subtransactions is maintained by the GTM. A simulated prepared state is used to record the state of a global subtransaction. During recovery, exclusive access is provided to the MDBS. The global system resubmits all subtransactions belonging to committed global transactions that have been recorded to be in a prepared state at the time of failure. This approach is a violation of system autonomy since a local DBMS is unavailable to local users until the global recovery manager completes recovery. However, it has been argued that this loss of autonomy may not be of concern in most cases, since many commercial DBMSs allow their respective administrators to control accesses to their database. The authority of the database administrator to choose the appropriate time to open the local database to the users can be employed in developing a protocol for the handshake between the global and local administrator upon recovery or startup of the local databases. The disadvantage of this solution lies in the fact that the local transactions are delayed until a handshake is completed.

The REDO Approach: In cases where the local DBMSs do not provide a prepare-to-commit interface, a global transaction may be aborted by the local DBMSs at any time, even after the GTM has voted to commit the transaction. If the global subtransaction is aborted by the local DBMS after the GTM has voted to commit the transaction, the server at the site at which the subtransaction was aborted submits a redo transaction consisting of all the writes performed by the subtransaction to the local DBMS for execution. The agent must maintain a server-log in which it logs the update of global subtransactions. If a redo transaction fails, it is resubmitted to the server until it commits. This approach requires that the schedules produced by the local DBMS be cascadeless. A transaction must read data that is already committed. However, the problem with the redo approach is that the LDBS views resubmitted transactions as new transactions; this could result in a non-serializable schedule.

Semantic Based Recovery: As noted earlier, transaction semantics can be used for concurrency control. The correctness criterion is that the execution of transactions results in a database state that is semantically consistent [25]. If a transaction spans over more than one site and the transaction aborted at some sites and committed at others, two directions could be taken for recovery. First, an aborted global subtransaction could be retried until it commits. The second

approach would be to use a compensating transaction to undo the effects from the committed global subtransactions. The compensating transaction will have to preserve database consistency as well.

Compensation achieves Semantic Atomicity. Specifically, within a history, the operation x is compensated by transaction x^{-1} if all future operations are executed in such a manner as if the operations x and x^{-1} never took place. Along with compensating transactions, compatibility can be specified in terms of the commutativity of the transactions. In that sense, a subsequent operation may have returned different values had strict serializability been followed. These values should be acceptable from the semantic point of view. Compensating transactions may not be applicable in all cases. For example, transactions leading to real world actions are not candidates for compensation. The idea of using compensating transactions was used in Sagas [26]. The concept of sagas is explained in the next section.

Semantic based transaction management is important in the development of future multidatabase systems. The knowledge of the environment for which the multidatabase is designed can be used effectively to resolve the problems encountered in transaction management. Advanced transaction models that use the semantics as well as the context can be beneficial. As a result, newer applications and environments can be integrated effectively into a multidatabase environment. These, in our opinion, deserve more attention. The next section is devoted towards the discussion of some more specific applications and the relevance of advanced transactions models in such an environment. **The O2PC Method:** The optimistic commit protocol (O2PC) is based on the concept of semantic atomicity [25]. When a transaction completes, the GTM sends "prepare" messages to the agents at each site. Upon reception of the prepare message, the agents optimistically try to commit their subtransaction. The result is reported to the GTM. If all the transactions are committed, the transaction is declared committed. Otherwise, the transaction is declared aborted and compensating transactions are executed at all sites where the subtransactions were committed. The problem here is that transactions that commit at some sites and abort at others may violate database integrity. Global inter-site integrity constraints are therefore not allowed. Transaction should be prevented from seeing the effects of failed (or compensated for) and successful subtransactions of global transactions.

4.5 Application Based and Advanced Transaction Management

Serializability requires that the execution of each transaction must appear to every other transaction as a single atomic step. This requirement could be too rigid and difficult to implement, particularly when dealing with CAD/CAM applications, long lived transactions, transactions in cooperative environments, engineering applications, and so forth. These applications reveal the need for extended transaction models, which exploit transaction context and semantics. In general, these models relax the ACID properties with weaker guarantees - nested transactions, Sagas [18].

4.5.1 Unconventional Transactions Types

Application specific extensions to transaction processing give rise to transactions having different requirements. Some examples are:

Long-lived transactions: These transactions usually consist of many steps and the transaction as a whole is characterized by its long duration. Aborting the transaction would mean a significant loss of work; hence, relaxed atomicity criteria are more appropriate. The transactions should support resumable computation, so that even in case of a system crash, a suspended transaction is not necessarily aborted. Challenges faced while handling long-lived transactions involve:

- Formulating isolation requirements such that short transactions executing concurrently are affected minimally, and
- Handling deadlocks due to the increase in the transaction duration.

Cooperating Transactions: Cooperating transactions form a transaction group in which partial results of the operations in progress can be shared. Such an environment is characterized by relaxed atomicity and relaxed isolation among transactions. To provide interactive control in a cooperating group of transactions, the transactions should be such that operations can be retracted, i.e., compensatable. The eventual requirement of correctness among cooperating transactions is to maintain group consistency and isolation.

4.5.2 Advanced Transaction Models

In an environment supporting long-lived transactions, it would be undesirable to lose a large amount of work because a transaction aborts or the system crashes. Sagas [26], based on compensating transactions and semantic atomicity, have been proposed to deal with long-lived transactions.

Sagas [26]: A saga is a long-lived transaction that consists of relatively independent subtransactions that can be interleaved in any way with other transactions. Associated with each subtransaction $T_1, T_2, ..., T_n$ defined in a saga are compensating transactions $C_1, C_2, ..., C_n$. The system then makes a guarantee that either the sequence $T_1, T_2, ..., T_n$ or the sequence T1, T2, ..., $T_j, C_1, C_2, ..., C_j$ for $1 \leq j \leq n$ will be executed. Thus, in a saga, all the subtransactions are completed or partial executions are undone by a compensating transaction. Sagas relax the property of isolation by allowing partial results to be visible to other transactions before completion. Sagas require that either all subtransactions successfully complete or none complete. Thus, sagas preserve atomicity and durability properties.

Sagas are applicable only in an environment when subtransactions are relatively independent and each subtransaction is compensatable. If the databases joining a MDBS are satisfying this condition, then this approach may be helpful in alleviating the atomic commitment problem that arises due to the blocking behavior of 2PC. An approach that utilizes sagas to maintain concurrency among global procedures in a federated DBMS environment is illustrated in [l].

Sagas do not support resumable computation because of their support for ACID properties of individual subtransactions, which in turn implies that all active transactions should be aborted on system crash. This raises the question, "what if forward recovery is desired"? The ConTract Model proposed by Reuter is for just such a purpose.

ConTract Model [51]: This model was proposed for dealing with long duration, complex computation transactions in non-standard applications like office automation, CAD, and manufacturing control. A ConTract is a predefined set of actions with an explicit specification of control flow among these actions. The main emphasis of this model is that a ConTract should be forward recoverable, implying that a computation should be resumable. To be able to do so, all state information, including database state, program variables of each step, and the global state of the ConTract, must be made recoverable. Similar to sagas, a ConTract is allowed to externalize its partial results before the ConTract actually completes, again relying on compensating transactions to undo effects of committed transactions.

Multilevel Transactions [60]: In this approach, the operations of a transaction are allowed to execute at different levels in the system. For example, a global transaction is executed at the global level and a local transaction is executed at the local level. Level specific conflict relations between operations provide an increasing abstraction from bottom to top level. In a multilevel transaction, operations that conflict at a lower level may commute at a higher level if the semantics of the operation are considered, making conflicts at the lower level pseudo conflicts. This model can be applied to MDBS by viewing global transactions at the topmost level (L2), the global subtransactions and the local transactions (LT) at the intermediate level (L1), and the data accesses of GSTs and LTs at the lowest level (LO). Using semantic and context information at L1 maintains the concurrency control.

The COSMOS project at ETH Zurich employs the open-nested transaction model, which is a generalization of the multilevel transaction model [59]. An open-nested transaction is a tree of subtransactions. Nodes at the same depth are at the same level of abstraction in the MDBS and edges represent caller-callee relationships. Concurrent executions in an open-nested model are semantically serializable. The open-nested transaction model has also been used in the implementation of VODAK [43], a distributed object-oriented database management system that allows transactions across heterogeneous and autonomous databases.

Flex Transaction Model [21]: The flex transaction model, used in the InterBase project at Purdue University, relaxes the atomicity and isolation properties of subtransactions to provide users with increased flexibility in specifying transactions. This model was proposed for multidatabase transaction management to provide an extended transaction model. The main features of this model are:

- It allows the user to give a set of acceptable states, i.e., it allows specification of a set of functionally equivalent subtransactions. This allows failure tolerance since it takes advantage of the fact that a given function can be executed in more that one way.

- Users can define the execution order of transactions in terms of internal dependencies and external dependencies of transactions.

- It allows the concept of mixed transactions. This allows compensatable and non- compensatable transactions to coexist.

- It allows the user to control the isolation granularity of a transaction through use of compensating transactions.

The above features contribute to the flexibility of transaction management. This extended model is particularly useful in a multidatabase model where local autonomy is of concern.

Reservable Transactions [20]: So far, we have concentrated on approaches where using the semantics of the transactions can increase concurrency - the effects of the transaction are rolled back using compensating transactions. This is an optimistic way of using the semantics of transactions. Another approach would be to use the data semantics to find out beforehand if the transaction can commit, without completely blocking the data that the transaction accesses. This phase is called the reservation phase for the transactions, and such transactions are called reservable transactions. This is more of a pessimistic approach to transaction management. A reservable transaction consists of a reservation phase followed by the actual transaction, provided the reservation phase is successful. Once the actual transaction completes, an unreservation phase is executed to undo the effects of reservation. A simple example would be booking a room in a hotel; the reservation phase would consist of making sure that a room is available followed by ensuring that any transaction that would make rooms unavailable is aborted.

Dynamic Restructuring of Transaction [33]: For long-duration transactions, it may be useful to reconfigure transactions while they are in progress. Two new operations, split transaction and join transaction, were introduced [33] for this purpose.

Definition 4 A split transaction divides an ongoing transaction into two new serializable transactions by dividing the actions and the resources between the new transactions. The splitting of a transaction generally results from new information about the dynamic access patterns of the transaction - for example; the transaction no longer needs resources. Therefore, one of the newer transactions can be committed in order to release its resources so that other transactions can access them.

Definition 5 A join transaction merges the ongoing work of two or more independent transactions as if it had always been a single transaction. This transaction model is especially useful in dealing with open-ended activities that have uncertain durations and unpredictable developments as they progress, i.e., VLSI design.

4.5.3 Replication

A replicated database is a distributed database in which multiple copies of some data items are stored at multiple sites. Replicated data has the following advantages

[6]:

1. Increased availability: by storing critical data at multiple sites, the DBS can operate even though some sites have failed.
2. Increased performance: since there are many copies of each data item, a transaction is more likely to find the data it needs close by, as compared to a single copy database.
3. Maximized system throughput: storing multiple copies of each data item at different sites has the potential to elevate the degree of concurrency and to improve the system's overall utilization [37,50].

However, these benefits come at the price of having to update all copies of each data item. Thus, reads may run faster at the expense of slower writes. A DBS managing a replicated database should behave like a DBS managing a one-copy (non replicated) database. The interleaved execution of concurrent transactions on the replicated database should be equivalent to a serial execution of those transactions on a one-copy database; this is often referred to as one- copy serializability (1SR). Thus, the goal of concurrency control in a replicated database should be to achieve one-copy serializability. This serves as the basis of consistency for replicated data.

Classification of replication techniques

Replication techniques can be classified into two categories based on the approach used to maintain the consistency of the replicated database [6]:

Write-All Approach: under this approach a DBS translates each read operation, r(x) into r(x_A) where x_A is any copy of data item x and it translates each write operation, w(x) into {$w(x_A)$, ..., w(x_Z)} where $\{x_A, ..., x_Z\}$ are all copies of x. Any serializable concurrency control algorithm is used to synchronize access to the copies. The problem with the write-all approach is that it assumes the ideal case where the sites never fail. However, in reality sites can fail and recover. Since there will be times when some copies of x are down, the DBS will not always be able to write into all the copies of x at the time it receives w(x). Thus, it would have to delay processing w(x) until it could write all copies. The more copies of x that exist, the higher the probability that one of them is down. As a result, more replication of data actually makes the system less available to update transactions, making the write-all approach unsatisfactory.

Write-All-Available Approach: this approach relaxes the constraint that requires all copies of each data item x to be written into by the DBS when a write operation, w(x) is received. Instead, a DBS should write into all available copies; it can ignore any copies that are down while still producing a serializable execution. The write-all-available approach solves the availability problem, but may lead to problems of correctness. There will be times when some copies of x do not reflect the most up-to-date value of x. A transaction that reads an out-of-date copy can create a non-1SR execution that is incorrect. For example, consider the following

execution history of three transactions, T_0, T_1 and T_2 issued at cites A, B, and C, respectively:

$$H_1 = w_{T_0}(x_A)w_{T_0}(x_B)w_{T_0}(y_C)c_{T_0}r_{T_1}(y_C)w_{T_1}(x_A)c_{T_1}r_{T_2}(x_B)w_{T_2}(y_C)c_{T_2}$$

T_2 reads the copy x_B of x from T_0, even though T_1 was the last transaction to write x before it. Thus, T_2 read an out-of-date copy. One reason why this could have occurred is because of a failure at site B. In keeping with the principle of the write-all-available approach, T_1 would ignore the failed copy. Thus, the execution history is non-1SR as a result of T_2 reading an out-of-date copy of x when B recovered. This problem can be resolved by preventing transactions from reading copies from sites that have failed and recovered until these copies have been brought up-to-date, though this alone is not enough.

Update propagation

When a transaction issues a write on a data item x, the DBS is responsible for eventually updating the set of copies of x. In eager replication, a write on a data item is distributed immediately to all the replicas [6]. With lazy replication, the DBS delays the distribution of writes to other copies until the transaction has terminated and is ready to commit. A DBS that uses lazy replication puts all writes destined to the same site in a single message. This tends to minimize the number of messages required to execute a transaction. In contrast, in eager replication, the DBS sends writes to replicated copies while the transaction executes, using in essence one message per write. In lazy replication, aborts often cost less than eager replication. In eager replication, when a transaction aborts, all the writes that have already been distributed to replicated copies are wasted and must be undone. Lazy replication defers the writes until the transaction terminates. As a result, the abortion of a transaction before its termination is less costly than eager replication.

In contrast, lazy replication may delay the commitment of a transaction more than eager replication. This is because the delayed write will have to be processed at commit time. An atomic commitment protocol (ACP) is used in order to assure consistency of replicated data at each site. The write processing delays the response to the vote request that the ACP employs. Eager replication can issue a vote right away since it has already performed the writes while the transaction was executing. Lazy replication also tends to delay the detection of conflicts between operations. If multiple transactions write to the same data item while using different copies, one or more of the transactions may be rejected at commit time and will have to be aborted after all the processing has been performed. This can be remedied by requiring the DBS to use the same copy of each data item (called the primary copy) to execute each transaction so that conflict may be detected earlier.

Replica control protocols

Replication of data should be transparent to the user. To achieve this, a synchronization layer called a Replica Control Protocol is employed. The protocol provides

a set of rules to regulate access (reading and writing) to replicas. These rules help to determine the actual value of a data item in the presence of conflicting replicas. A number of replica control protocols have been developed and addressed in the literature [37,50]. These can be categorized into three groups: Primary-Copy Method, Quorum Consensus Method and Available-Copies Method. Other methods proposed in the literature are extensions to these approaches.

Primary-Copy Method: this method assumes that all replicated data in the system have a predefined order. The copy directly maintained by the data owner, or primary copy, takes the first place in the order. The other replicas are secondary copies. To maintain 1SR, write operations are always performed on the primary copy first and then propagated to the other replicas either synchronously or asynchronously while a read operation can choose from any available copy. The method is also called the Read-One-Write-All method. If a transaction reads multiple data copies and eventually finds out that the updates are still in propagation and some replicas are not fully up-to-date, the transaction will either be aborted or directed to the primary copy of the data item to ensure the correctness of the data. In case of a failure in the node hosting the primary copy, the copy with the next highest order takes over and performs as the primary copy. The primary copy method has the advantage of being easy to implement. It is straightforward and capable of ensuring 1SR but is vulnerable to communication failures. The overhead of propagating updates to all copies may eventually become a bottleneck, particularly when the number of replicas is large.

Quorum-Consensus Method: in this method an operation is allowed to execute if it can obtain voting permission from a group, or quorum, of nodes containing the replicas of the targeted data item. Consequently, this method is also referred to as the Voting method in the literature. In general, two quorum sets must be defined in a system, one for reads and the other for writes. A read operation should get permission from a read quorum group and a write operation should get permission from a write quorum group. Quorums are formed according to the following rules:

1. Any two write quorum groups must have a common member, which ensures that no two writes can execute concurrently.
2. A read quorum and a write quorum group for a data item must have at least one node in common. This would ensure that conflicting read and write operations do not access a specific replica of the item concurrently.

Usually when inconsistencies are detected within a quorum during voting, reconciliation processes will start to synchronize the values of copies. A transaction that fails the voting test will either be delayed after a reconciliation phase or aborted. This method is free from communication failures because of redundant data access, but the performance is relatively poor due to high communication overhead. The assignment of quorums lacks flexibility and is difficult to be optimized automatically. A new copy of the data cannot join the system until the quorums are rearranged. This effectively limits the scalability of the system.

Available-Copies Method: this is also known as optimistic replication. All available copies of a data item are updated during a write operation while reads can access any available replica. This method does not limit data availability for maintaining data consistency. Transactions are allowed to execute even during network partitioning when some of the copies may be unavailable and thus inconsistent. When the sites recover, the inconsistencies will be detected and some convergence operations will be launched to synchronize the data values. Transactions that accessed inconsistent data have to be undone. Single copy serializability is abandoned and other mechanisms such as timestamp or version numbers are used to enforce data convergence. The available copies method can achieve high performance of both read and write operations. However, the reconciliation and validation tests require additional message communication. When the number of data copies or the time period of partition increases, efficiency is compromised due to the number of data convergence operations and increase in communication.

Multidatabase replication issues

The nature of the multidatabase makes the issue of providing and maintaining replicas of data items a not trivial issue. Two identifying features characterize a multidatabase: heterogeneity and local autonomy. Traditionally, replication has been designed for and implemented on homogenous distributed systems. These systems co-operate with each other, follow the same locking policies, concurrency control protocols and recovery algorithms, and have similar system architectures and, more importantly, are not autonomous. Replica control protocols in multidatabase should be designed to work in the presence of heterogeneous, autonomous nodes. They should function in the presence of various transaction processing and locking schemes.

Replication often requires that the participating sites implement a global atomic commitment protocol (ACP) in order to ensure the consistency of the replicated data. This implies that the databases at the sites are not autonomous. In contrast, the local database at the local sites in a multidatabase is autonomous. Providing a global atomic commitment protocol in the multidatabase is practically impossible without violating the local autonomy of the databases. Further more, local autonomy implies that the local user may read and write to a replicated data item x at the local site without the global layer being aware of the transaction. Such an execution, though correct at the local site, is a violation of global consistency. A simple solution may disallow local users from accessing the replicas, but this again is a violation of local autonomy.

In a multidatabase, each local site is free to join or leave the MDBS at any time. Thus, the replica control protocol should be designed to operate with a minimum of disruption when a node leaves the MDBS. This is similar to the failure of a node in a traditional replicated system with the difference that the local node may or may not rejoin the MDBS.

Replication should be transparent to the database user. The approach used in

the design of the database will affect how transparent replication is to the user. The global schema approach is well suited to replicating whereas the multidatabase language approach is not [10]. The multidatabase language approach provides users with control over replication instead. Thus, the user is aware of replication, and has the responsibility of managing replicas of data items.

Consistency of data is often maintained via one-copy serializability in replicated systems. This has been accepted as the de facto standard. However, maintaining serializability without global control, as is the norm in multidatabases, is difficult and costly to implement. This difficulty has led to weaker notions of correctness and consistency in multidatabases, which could have a significant effect on replication control protocols such that they might have to be redesigned to accommodate the different notions of consistency.

MDAS replication issues

In the previous subsection we discussed the issues that affect replication within a multidatabase environment. This section discusses the issues that affect replication in a MDAS environment. The key issues in a multidatabase are local autonomy and heterogeneity; the MDAS, in addition, introduces the issues of frequent disconnection, limited wireless bandwidth, higher error rates, limited computational resource (CPU, memory, storage) and limited battery life.

Bringing the data closer to the mobile client would result in quicker response times, particularly, due to the bandwidth limitations experienced by the mobile user. Replicating data at the mobile unit would also allow the mobile client to continue processing, to some extent, in the likely event of a disconnection (isolation). These are some of the advantages that make replication a very attractive proposition for the MDAS. However, any attempt at maintaining replicas must first of all determine how to overcome the inherent limitations of the mobile computing environment.

Replica control protocols differ in the distribution of replication function between the mobile client and the server, and the allocation of update capabilities at the client. Some systems allow updates of offline replicas while others do not. Data can be replicated at three levels in the mobile environment. The first level is at the source system, where multiple copies are maintained at different base stations. Dynamic replication can be used at this level to redistribute data to get closer to the access activities. Classical replication protocols are sufficient to handle this. The second level involves caching data into entities referred to as data access agents (DAA) base stations. When a user first requests data, the DAA holds it in its cache. As the user moves, the DAA caches replicate to DAAs at other base stations. The third level involves caching the DAA's cache at the mobile unit. Maintaining the consistency of the cache and the replicas at the three levels is the focus of replication management [50].

In the previous subsection we pointed out that in order to ensure correctness, the multidatabase must implement some form of atomic commitment protocol. In reality, this is a difficult task while preserving the autonomy of the local database.

Frequent disconnection and mobility often results in long-lived transactions, complicating the matter further. In the MDAS environment, this would allow a mobile unit to hold on to locks for a relatively long time if locking schemes are being used.

Replication in traditional databases has mainly tried to maintain the 1SR property. The mobility of the mobile client, however, makes maintaining global 1SR a very costly event. The high overhead cost is due to the fact that mobile transactions tend to share states and results of partial transaction execution as the mobile unit migrates from one MSS to another. Frequent disconnection also implies that replicas will often be inconsistent, particularly, if the mobile client is allowed to perform operations on replicated data items while in the disconnected mode. The mobile transaction may have to be aborted upon reconnection and any other transactions that had accessed the data item may have to be rolled back and undone, or compensated for in order to maintain consistency of the global database. Thus, maintaining the 1SR property may not be suitable within the MDAS environment, as the cost may prove too prohibitive.

An overview of the how location information, power restrictions and orderly connections/disconnections influence the various replica control protocols can be found in [23].

Replication solutions in multidatabases

Two protocols for replicated data management are presented in [32]. These protocols employ deferred propagation for local applications and immediate propagation for global applications. The first protocol ensures replication consistency (with regard to 1SR) through the use of a global certification protocol, which verifies the consistency of replicated copies accessed by both local and global applications. The second protocol is an extension of the first and uses propagation lock on primary copy sites. The propagation locks permit the consistency criterion to be performed locally for single site query applications, thereby improving the performance of transaction processing. The protocols are describe as follows:

Two-Phase Certification Protocol: In this protocol, a global transaction is submitted to the GTM, where it is decomposed and sent to the local sites. A primary-update subtransaction is submitted locally. After the primary-update subtransaction finishes its execution, it is immediately committed at the local DBMS. A propagation transaction is then formed and sent to the GTM. The GTM monitors and executes the propagation transaction just like a regular global transaction. The approach has the advantage that it maintains replication consistency without violating local autonomy. The protocol, however, requires that an application, which accesses copies that logically belong to different sites, be processed as a global transaction. A consequence of this is that it assumes that each LDBS provides a visible prepared-to-commit state for its transactions. This implies that a two-phase commit protocol must be used to control commitment. Even a single-site query that only reads non-primary copies must be treated as a global transaction. Otherwise, 1SR cannot be guaranteed globally. The ability to process single- site queries as

local transactions, without the involvement of the GTM is critical to data availability and system performance. This protocol relaxes the requirement that single- site queries be processed as global transactions by relaxing the control of the GTM over single-site queries through the use of a weak consistency requirement [32].

Propagation Lock Protocol: this protocol is an extension of the 2P Certification Protocol. It provides an alternative approach to processing single-site queries as local transactions. Instead of relaxing the consistency requirement, this approach adopts a propagation lock to prevent possible inconsistent access. The propagation lock blocks the execution of single-site queries at the primary copy site until the conclusion of other propagation transactions. The lock is set on the primary copy site when a primary-update subtransaction is submitted to the site; it is released after the propagation transaction globally commits at remote sites. The server layer at the primary copy site is responsible for granting and releasing the locks. Two propagation locks are compatible and can be granted at the same time on the primary-copy site. The lock blocks only single-site query transactions, leaving other transactions unaffected. The GTM monitors the executions of propagation transactions and will inform the local server of the commitment of a propagation transaction. A single-site query can be submitted and committed as a local transaction if no propagation lock is set on the local site. If a propagation lock is set between the submission and the commitment of a single site query, it will be aborted and rerun after the lock is released. Thus, the single-site query is executed and committed locally without any coordination by the GTM. The propagation lock approach ensures 1SR; it does not require global control on the execution of a primary-update transaction subtransaction, as the lock is set only on the primary copy site where the primary-update transaction is executed. The approach may, however, delay some single-site queries on primary copy sites. It also imposes some restriction on local concurrency control, in the form of rigorous schedules. The assumption of rigorous schedules is, in general, practically acceptable, since most commercial DBMS products use strict 2PL [32].

4.5.4 Replication Solutions in MDAS

This section discusses two replication control protocols for the MDAS, Virtual Primary Copy and Adjustable Replication Protocol.

Virtual Primary Copy (VPC) Protocol: this was proposed in [23] as a modification to the primary copy method adapted to suit the mobile environment with its distinctive features including frequent disconnections, power limitations, and mobility. It is based on a number of assumptions:

1. The Home Base Node (HBN) - MSS - knows the locations of the VPC of the Mobile Primary Copy (MPC) that has connected to it.

2. Whenever the MPC connects to any base node, the base node it connects to is able to contact the VPC.

3. Whenever the MPC connects to another HBN and executes some transaction from the new base node, then the base node to which it is connected becomes the new VPC and HBN has a pointer to this VPC.

At any instant in time, there exists one VPC for every MPC. The read requests for the MPC are handled in the same way as for the primary copy method. The actual location of a primary copy is transparent to global transactions, which access the VPC. The consistency of the VPC is maintained by the HBN. The proposed VPC method differs from a classic primary copy method in that the mobility of hosts is considered, disconnection of mobile hosts is handled and monitored by the HBN, and a multilayered approach is adapted by HBN, which is transparent to other sites.

Adjustable Replication Protocol: this is an optimistic replication protocol proposed in [37] that either rolls back an incorrectly committed transaction or sends a warning message to a user that retrieves outdated data. The local transactions of a site hosting a non-primary copy replica are limited to reading the secondary copies of the data in order to avoid problems associated with arbitrary data updates. All updates to secondary copies are issued as global transactions. Update requests submitted by local users are redirected to, and processed by, the global system. The protocol assumes the MDAS is organized using the Summary Schemas Model (SSM) structure. Consistency is maintained at the global level by a new locking structure known as the Horizontal V-lock, an extension of the V-lock algorithm that has the capability of horizontally directing subtransactions in the SSM hierarchy. Replicated data is stored in a fixed host. According to the application domain requirements and the characteristics of the data, a consistency requirement (CR) table records the consistency degree demanded by each data as is set by the data owner or the MDAS system replication manager. A simple cache database (CD) is reserved and used to hold the latest replicated data value at the MSS. The data cached in the CD is refreshed whenever the data is updated. While offline, operations are performed based on the data cached in the CD and transactions that are processed are logged in the history. The storage space of the CD is adjusted in accordance to the resources available at the MSS. On reconnection, a reconciliation process is launched to eliminate the data delusion caused by offline processing and rollback/recovery of the falsely committed operations will be carried out.

4.6 Experiments with V-Locking Algorithm

As noted before, within the MDAS environment a proper transaction model should address issues such as the limited network bandwidth and frequent disconnections. The so-called V-locking model is a hierarchical concurrency control algorithm that uses global locking tables created with semantic information contained within the hierarchy.

The semantic information contained in the summary schemas is used to maintain global locking tables. Since each summary schema node contains a semantic content of its children schemas, the "data" item being locked is reflected exactly or as a

hypernym term in the summary schema of the GTM. The locking tables can be used in an aggressive manner where the information is used only to detect potential global deadlocks. A more conservative approach can be used where the operations in a transaction are actually delayed at the GTM until a global lock request is granted. In either case, the global locking table is used to create a global wait-for-graph, which is subsequently used to detect and resolve potential global deadlocks.

The accuracy of the "waiting information" contained in the graph is dependent upon the amount of communication overhead that is required. The proposed algorithm can dynamically adjust the frequency of the communications (acknowledgment signals) between the GTM and local sites based on the network traffic and/or a threshold value. The number of acknowledgments that are performed varies from one per operation to only a single acknowledgment of the final commit/abort of the transaction. Naturally, the decrease in communication between the local and global systems comes at the expense of an increase in the number of potential false aborts.

The algorithm handles three different levels of communication: 1) each operation in the transaction is individually acknowledged, 2) write operations are only acknowledged, and 3) only the commit or abort of the transaction is acknowledged. For the first case, based upon the semantic contents the summary schema node, an edge inserted into the wait-for-graph is marked as being an exact or imprecise data item. For each acknowledgement signal received the corresponding edge in the graph is marked as exact. In the 2nd case, where each write operation generates an acknowledgement signal, for each signal only the edges preceding the last known acknowledgement are marked as being exact. Other edges that have been submitted but have not been acknowledged are marked as pending. As in the previous two cases, in the 3rd case, the edges are marked as representing exact or imprecise data. However, all edges are marked as pending until the commit or abort signal is received. Keeping the information about the data and status of the acknowledgement signals enables us to detect cycles in the wait-for-graph.

The algorithm detects cycles in the wait-for-graph based on the depth first search (DFS) policy. The graph is checked for cycles after a time threshold for each transaction. For all of the transactions involved in a cycle, if the exact data items are known and all of the acknowledgements have been received, then a deadlock is precisely detected and broken. When imprecise data items are present within a cycle, the algorithm will consider the cycle a deadlock only after a longer time threshold has passed. Similarly, a pending acknowledgement of a transaction is only used to break a deadlock in a cycle after an even longer time threshold has passed. The time thresholds can be selected and adjusted dynamically to prevent as many false deadlocks as possible.

A potential deadlock situation may also occur due to the presence of indirect conflicts. By adding site information to the global locking tables, an implied wait-for-graph could be constructed using technique similar to the potential conflict graph algorithm [7]. A potential wait- for-graph is a directed graph with transactions as nodes. The edges are inserted between two transactions for each site where there are both active and waiting transactions. The edges are then removed when

a transaction aborts or commits. A cycle in the graph indicates the possibility that a deadlock has occurred.

4.6.1 Simulation Studies

The performance of the proposed algorithm was evaluated through a simulator written in C++ using CSIM. The simulator measures performance in terms of global transaction throughput, response time, and CPU, disk I/O, and network utilization. In addition, the simulator was extended to compare and contrast the behavior of our algorithm against the site-graph, potential conflict graph, and the forced conflict algorithms.

The MDAS consists of both local and global components. The local component is comprised of local DBMS systems, each performing local transactions outside the control of the MDAS. The global component consists of the hierarchical global structure, performing global transactions executing under the control of the MDAS. There are a fixed number of active global transactions present in the system at any given time. An active transaction is defined as being in the active, CPU, I/O, communication, or restart queue. A global transaction is first generated, and subsequently enters the active queue. The global scheduler acquires the necessary global virtual locks, and processes the operation. The operation(s) then uses the CPU and I/O resources, and communicates the operation(s) to the local system based upon the available bandwidth. When acknowledgements or a commit/abort signals are received, from the local site, the algorithm determines if the transaction should proceed, commit, or abort. If a global commit is possible, then a new global transaction is generated and placed in the ready queue. However, if a deadlock is detected, or an abort message is received from a local site, then the transaction is aborted at all sites and the global transaction is placed in the restart queue. After a specified time elapsed, the transaction is again placed on the active queue.

Each local site has a fixed number of active local transactions comprising of both local transactions and global sub-transactions. The local system does not differentiate between the two types. An active transaction is defined as being in the active, CPU, I/O, communication, blocked, or restart queue. Transactions enter the active queue and are subsequently scheduled by acquiring the necessary lock on a data item. If the lock is granted, the operation proceeds through the CPU and I/O queue, and for global sub-transactions, is communicated back to the GLS. The acknowledgement for these transactions is communicated back based upon the available communication bandwidth. If a lock is not granted, the system checks for deadlocks and will either place the transaction in the blocked queue, or the restart queue. For local transactions, it goes into the restart queue if it is aborted, and subsequently it will be restarted later. Upon a commit, a new local transaction is generated and placed in the ready queue. For global sub- transactions, an abort or commit signal is communicated back to the GLS and sub-transaction terminates.

4.6.2 System Parameters

The underlying global information sharing process is composed of ten local sites. The size of the local databases at each site can be varied, and has a direct effect on the overall performance of the system. The global workload consists of randomly generated global queries, spanning over a random number of sites. Each operation of a sub-transaction (read, write, commit, or abort) may require data and/or acknowledgements to be sent from the local DBMS. The frequency of messages depends upon the quality of the network link. In order to determine the effectiveness of the proposed algorithm, several parameters are varied for different simulation runs.

The local systems perform two different types of transactions, local and global. Global sub-transactions are submitted to the local DBMS and appear as a local transaction. Local transactions are generated at the local sites and consist of a random number of read/write operations. The number of local transactions, which can be varied, affects the performance of the global system. In addition, the local system may abort a transaction, global or local, at any time. If a global sub-transaction is aborted locally, it is communicated to the global system and the global transaction is aborted at all sites.

4.6.3 Simulation Results

The performance of the algorithm (V-Lock) is evaluated based on performance metrics such as, the number of completed global transactions, the average response time, as well as the communication utilization at each local site. In addition, the simulator was extended to compare and contrast the proposed algorithm against the potential conflict graph method [7], site graph method [30], and the forced conflict method [9]. Figure 4.9 shows the results. As can be concluded, the V-Lock algorithm completes the most transactions. This result is consistent with the fact that the V-Lock algorithm is better able to detect global conflicts and thus achieves higher concurrency than the other algorithms. As can be seen, the maximum occurs at a multi- programming level of approximately equal to ten. As expected, as the number of concurrent global transactions increases, the number of completed global transactions decreases due to the increase in the number of conflicts.

Figure 4.10 shows the relationship between the global throughput and the number of sites in the MDAS environment in which the number of sites was varied from 10 to 40. The throughput decreases as the number of sites is increased due to the data fragmentation across more sites, and hence increasing the likelihood of conflicts in the system.

The simulator also measured the percentage of completed transactions during a certain period of time for the V-Lock, PCG, forced conflict, and site-graph schemes. Figure 4.11 shows the results. In general, for all schemes, the number of completed transactions decreases as the number of concurrent transactions increases, due to more conflicts among the transactions. However, the performance of both the forced conflict and site-graph algorithms decreases at a faster rate. This is due to the

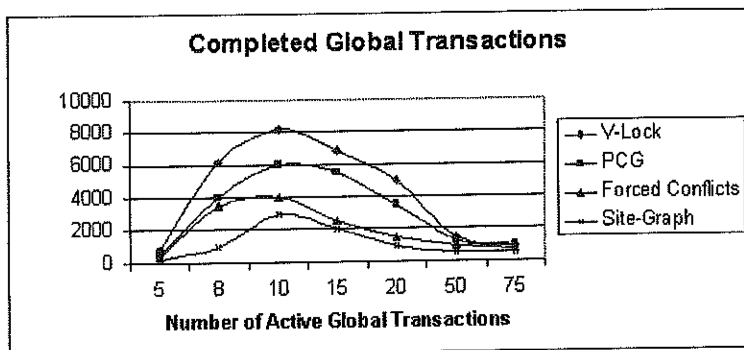


Figure 4.9. Comparative analysis of different Concurrency Control Algorithms.

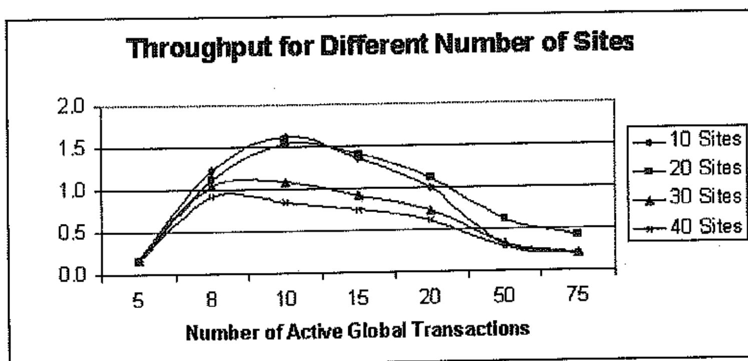


Figure 4.10. Global Throughput Varying the Number of Local Sites.

increase in the number of false aborts detected by these algorithms.

Finally, on a separate simulation runs, the simulator measured, and compared and contrasted the response time for various simulation runs. Figure 4.12 shows the results. The two locking algorithms have a much better response time than the forced-conflict and site-graph algorithms. The V-locking algorithm has the best response time, and performs better than PCG, particularly for a large number of users. As expected, as the number of concurrent users increases, the response time increases.

4.7 Conclusion

The need to maintain local autonomy is the distinguishing factor in transaction management for multidatabase systems. The problems associated with transaction management in multidatabases have been examined, and solutions proposed in the literature to deal with the concurrency control and recovery problem in multi-

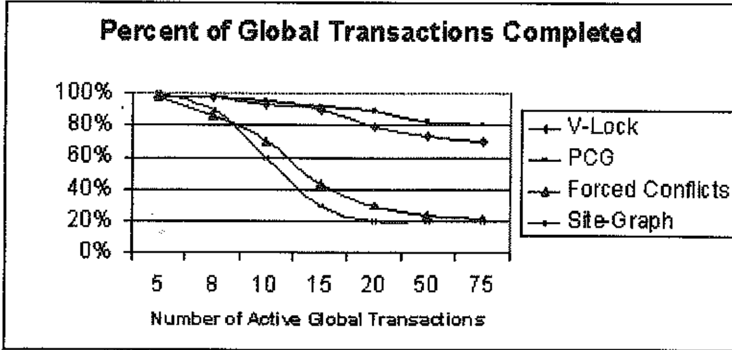


Figure 4.11. Comparison of the Percent of Completed Global Transactions.

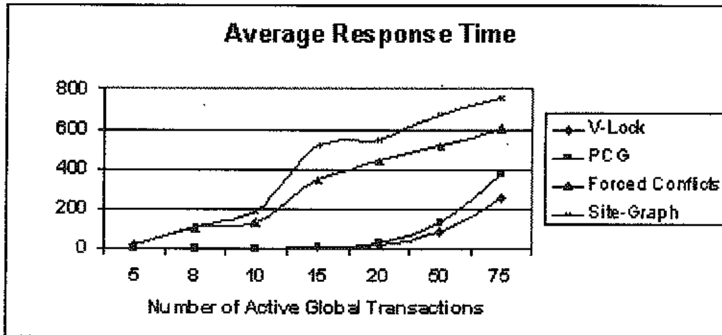


Figure 4.12. Average Response Time.

database systems have been studied. With the emergence of new applications for E-commerce and advanced transaction models, many interesting issues arise. The challenge is to combine these models with traditional transaction management models. A multidatabase system is primarily designed to integrate different and existing database systems; the necessity of combining different transaction models is hence very relevant for multidatabase systems of the future. The goal is to develop extensible transaction management schemes that meet the application specific needs of different local database systems in the most efficient manner.

As designers try to meet these needs, a number of decisions on the properties of multidatabase systems will have to be made. Since the most severe restriction is the autonomy of the local databases, it is likely that designers will look there for relief. One interesting approach is to allow the local database to retain full control over what and how transactions are performed on the data, but requires local database systems to provide information to the multidatabase system.

A variety of information can be used to support transaction management. The

simplest type of useful information is for the local database system to supply the multidatabase system with a complete local scheme. More difficult to obtain, but perhaps the most useful information, is to have the local database system provide the global transaction system with information on the transactions performed on the local database. The typical multidatabase system requires some kind of "global wrapper" on each system supporting local databases. By including a scheduler in the global wrapper, it can make use of the information on the transactions to verify the global order. Once the global order is verified, this information can be communicated to the global transaction scheduler. This information can then be used to ensure that global serializability is eventually maintained. By incorporating such approaches, future transaction management systems for multidatabases will become more practical.

Effective E-commerce technologies are in their formative stage. As E-commerce moves from a largely business-to-business model to include more retail selling channels, the demand for more effective multidatabases systems and mobile data access systems will proliferate. The problems of effective multidatabase systems need to be resolved to allow the development of electronic markets.

4.8 Bibliography

[1] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency Control and Recovery for Global Procedures in Federated Database System. *Q. Bulletin of the Computer Society of IEEE Technical Committee on Data Eng.*, 110(3), September 1987.

[2] R. Alonso and H. F. Korth. Database System Issues in Nomadic Computing. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 388 - 392, 1993.

[3] K. Barker and M. T. Ozsu. Reliable Transaction Execution in Multidatabase Systems. In *Proceedings of First International Workshop on Interoperability in Multidatabase Systems*, April 1991.

[4] K. Barker. Transaction Management on Multidatabase Systems. Ph.D. thesis, Department of Computing Science, University of Alberta, 1990.

[5] D. A. Bell. *Distributed Database Systems.* Addison-Wesley Publishing Company, 1992.

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

[7] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschtz. On Rigorous Transaction Scheduling. *IEEE Transactions on Software Engineering*, 17(9): 954-960, September 1991.

[8] Y. Breitbart and A. Silberschtz. Multidatabase Update Issues. In *Proceedings of ACM SIGMOD Intl. Conference on Management of Data*, pp. 135-142, 1988.

[9] Y. Breitbart, H. Garcia-Molina and A. Silberschtz. Overview of Multidatabase Transaction Management. *VLDB*, 1(2): 181-239, 1992.

[10] M. W. Bright, A. R. Hurson and S. H. Pakzad. A Taxonomy and Current Issues in Multidatabase Systems. *IEEE Computer*, 25(3): 50-60, 1992.

[11] M. Bright, A. R. Hurson and S. Pakzad. Automated Resolution of Semantic Heterogeneity in Multidatabases. *ACM Trans. On Databases*, 19(2), pp. 212-253, 1994.

[12] S. Ceri and G. Pelagatti. *Distributed Databases Principles and Systems.* McGraw-Hill Book Company, 1984.

[13] P. K. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2): 194-203, 1990.

[14] P. Chrysanthis. Transaction Processing in Mobile Computing Environment. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pp. 77 -82, 1993.

[15] R. A. Dirckze and L. Gruenwal. Nomadic Transaction Management. *IEEE Potentials* Volume: 17(2), pp. 31 - 33, April - May 1998.

[16] W. Du and A. Elmagarmid. Quasi Serializability: A Correctness Criterion for Global Concurrency Control in InterBase. In *Proc. of 15th Int. Conf. on Very Large Databases*, pp. 347-355, 1992.

[17] M. H. Dunham, A. Helal and S. Balakrishnan. A Mobile Transaction Model That Captures Both the Data and Movement Behavior. *Mobile Network Applications* 2, 2, pp. 149 -162, October 1997.

[18] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.)

[19] A. Elmagarmid and W. Du. A Paradigm for Concurrency Control in *Heterogeneous Distributed Database Systems.* In Proc. Sixth Intl. Conf. on Data Engineering, pp. 37-46, IEEE, 1990.

[20] A. Elmagarmid, J. Jing, J. G. Mullen, and J. Sharif-Askary. Reservable Transactions: An Approach for Reliable Multidatabase Transaction Management. Technical Report SERC- TR- 1 14-P. Software Engineering Research Centre, April 1992.

[21] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz: A Multidatabase Transaction Model for InterBase. *VLDB*, pages 507-5 18, 1990.

[22] A. A. Farrag and M. T. Ozsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4): 503-525, December 1989.

[23] M. Faiz and A. Zaslavsky. Database Replica Management Strategies in Multidatabase Systems with Mobile Hosts.

[24] G. H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *IEEE Computer* Volume: 27(4), pp. 38-47, April 1994.

[25] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database System*, 8(2): 186:2 13, 1983.

[26] H. Garcia-Molina. Sagas. In *Proc. of ACM-SIGMOD 1987 Intl. Conf. On Management of Data*, pp. 249-259, 1987.

[27] H. Garcia-Molina. Node Autonomy in Distributed Systems. In *Proc. Intl. Symp. on Databases in Parallel and Distributed Systems*, pp. 158-166. IEEE, 1988.

[28] H Garcia-Molina. Global Consistency Constraints Considered Harmful for Heterogeneous Database Systems. In *Proc. of First International Workshop on Interoperability in Multidatabase Systems*, pp. 248-250, 1991.

[29] D. Geogakopoulos. Multidatabase Recoverability and Recovery. In *Proceedings of First International Workshop on Interoperability in Multidatabase Systems*, April 1991.

[30] D. Geogakopoulos, M. Rusinkiewiez and A. Sheth. Using Tickets to Enforce the Serializability of Multidatabase Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 6(l): 166-180, 1994.

[31] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[32] J. Jing, W. Du, A. Elmagarmind and O. Bukhres. Maintaining Consistency of Replicated Data in Multidatabase Systems. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pp: 552 -559, 1994.

[33] G. E. Kaiser and C. Pu. Dynamic Restructuring of Transactions. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, pp. 265-295, 1992.

[34] J. J. Kistler and M. Satyanarayanam. Disconnected Operation in Coda File System. *ACM Trans. Computer Systems*, Vol. 10, No. 1, pp. 3-25, Feb. 1992.

[35] R. Krishnamurthy, W. Litwin and W. Kent. Interoperability of Heterogeneous Databases with Schematic Discrepancies. In *Proc. First Intl. Workshop on Interoperability in Multidatabase Systems*, pp. 144-152. IEEE, 1991.

[36] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM TODS,* 6(2), 1981.

[37] C. Li. Replication Protocol for Mobile Data Access System - An Approach Under Summary Schemas Model. MS Masters Thesis, Department of Computer Science and Engineering, The Pennsylvania State University, May 2000.

[38] J. B. Lim, A. R. Hurson, K. M. Kavi. Concurrent Data Access in a Mobile Heterogeneous System. *Proceedings of the 32nd Annual Hawaii International Conf. on System Sciences.* 1999.

[39] W. Litwin and A. Abdellatif. Multidatabase Interoperability. *IEEE Computer*, 19(12): 10- 18, 1986.

[40] S. K. Madria and B. Bhargava. A Transaction Model for Mobile Computing. In *Proc. of the International Database Engineering and Applications Symposium*, pp. 92 -102, 1998.

[41] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Non-Serializable Executions in Heterogeneous Distributed Database Systems. In *Proc. of First International Conference on Parallel and Distributed Information Systems*, pp. 245-252, 1991.

[42] J. G. Mullen, A. K. Elmagarmid, Won Kim, J. Sharif-Askary. On the Impossibility of Atomic Commitment in Multidatabase Systems. Technical Report SERC-TR- 11 3-P, Software Engineering Research Centre, April 1992.

[43] P. Muth, J. Veijalainen, E. J. Neuhold. Extending Multi-Level Transactions for Heterogeneous and Autonomous Database Systems. GMD Technical Report No. 739, Santa Augustin, March 1993.

[44] E. Pitoura, and B. Bhargava. Revising Transaction Concepts for Mobile Computing. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pp. 164 - 168, 1995.

[45] C. Pu, A. Leff and S. F. Chen. Heterogeneous and Autonomous Transaction Processing. *IEEE Computer*, 24(12): 64-72, 1991.

[46] C. Pu. Superdatabases for Composition of Heterogeneous Databases. In *Proceedings of Fourth International Conference on Data Eng.*, pp. 548-555, 1988.

[47] C. Pu and Shu-Wie Chen. ACID Properties Need Fast Relief: Relaxing Consistency Using Epsilon Serializability. In *Proceedings of Fifth International Workshop on High Performance Transaction Systems*, September 1993.

[48] C. Pu, G. Kaiser and N. Hutchinson. Split-Transactions for Open-ended Activities. In *Proc. of the 14th VLDB Conference*, 1998.

[49] K. Ramamritham and C. Pu. A Formal Characterization of Epsilon Serializability. *IEEE Trans. on Knowledge and Data Engineering*, December 1995.

[50] R. Ramasubramanian. A Survey of Replication Issues and Strategies in Mobile and Multidatabase Environments. M Eng. Technical Paper, Department of Computer Science and Engineering, The Pennsylvania State University, August 1998.

[51] A. Reuter and H. Wachter. The ConTract Model. *Data Engineering Bulletin*, 14(1):39-43, 1991.

[52] B. Roberto and S. Silvio. Deadlock Detection in Multidatabase Systems: a Performance Analysis. Technical Report RR-2668, INRIA, The French National Institute for Research in Computer Science and Control, September 1995.

[53] M. Satyanarayanam. Mobile Information Access. *IEEE Personal Communications*, pp. 26-33, February 1996.

[54] M. Satyanarayanam. Mobile Computing. *IEEE Computer*, Volume 26 9, pp. 81-82, September 1993.

[55] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed Heterogeneous, and Autonomous Databases. *IEEE Computer*, 22(3): 183-236, 1990.

[56] N. Soparkar H. F. Korth and A. Silberschatz. Failure Resilient Transaction Management in Multidatabases. *IEEE Computer*, 24(12): 28-36, 1991.

[57] G. D. Walborn and P. K. Chrysanthis. Supporting Semantics-Based Transaction Processing in Mobile Database Applications. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, pp. 31 -40, 1995.

[58] G. Weikum and Hans-Jvrg Schek. Multi-Level Transactions and Open-Nested Transactions. *Data Engineering Bulletin*, 14(1): 60-64, 1991.

[59] G. Weikum, A. Deacon, W. Schaad, H.-J. Schek. Open Nested Transactions in Federated Database Systems. *IEEE Data Engineering Bulletin*, Vol. 16, No. 2, June 1993.

[60] A. Wolski and J. Vijalainen. 2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabases. In *Proc. of PARAIBASE-90 Intl. Conf. on Databases, Parallel Architectures, and Their Applications*, March 1990.

[61] L. H. Yeo and A. Zaslavsky. Submission of Transaction from Mobile Workstations in a Cooperative Multidatabase Processing Environment. In *Proc. of the 14th International Conference on Distributed Computing Systems*, pp. 372 - 379, 1994.

[62] V. Zwass. Structure and Macro-Level Impacts of Electronic Commerce: From Technological Infrastructure to Electronic Marketplaces. Foundations of Information Systems: E-Commerce Paper. [http://www.mhhe.com/business/mis/zwass/ecpaper.html]