TRANSACTION EXECUTION
IN
GRID DATABASE

By
Micheline ABBOUD

A thesis
Submitted for partial fulfillment
of the requirements for the degree of
Masters in Computer Information Systems

Department of Computer Science
Faculty of Natural and Applied Sciences
Notre Dame University

April 2005
CONCURRENT TRANSACTION EXECUTION
IN
GRID DATABASE

By
Micheline P. ABOUD

Approved

Dr. Marie Khair: Associate Professor of Computer Science
Advisor

Dr. Hoda Malaouf: Chairperson of Computer Science
Member of the Committee

Dr. Mario Missakian: Assistant Professor of Computer Science
Member of the Committee

Dr. Malhab Keyrouz: Associate Professor of Mathematics
Member of the Committee

Date of Thesis Defense: April 18, 2005
Acknowledgements

Now that I have come to this ultimate point in preparing for my Master’s thesis, I can take a deep breath and think about all the people here and there who have been helpful in my completing this project.

First of all, I would like to thank my supervisor, Dr. Marie Khair, without whose support, guidance and counsel this research would not have seen the light of day.

Heart-felt thanks to my parents and my fiancé for their support and encouragement, which made a world of difference to me and never failed to pep me up when I needed it most.

Special thanks are due to my employer – Soft Solutions – for graciously giving me the opportunity to continue my studies and bearing with me all this time, and for supporting employees in their pursuit of higher educational goals.

Last but not least, warm thanks to my dearest university NDU, its staff members and professors whose patience and support made the whole Master’s endeavor possible.
Abstract

The Grid initiative provides an infrastructure for distributed computation among widely distributed high-performance computers. This will allow for exchanging and processing very large amounts of data. There are a huge number of applications that are geographically distributed such as multi-point radio facility for astrophysics, space physics, atmospheric physics, and radio research, utilizing very high performance Grid Computing.

It is expected that similar applications to become important in commercial settings, initially for scientific and technical computing applications and then for commercial distributed computing applications, including enterprise application integration and business to business (B2B) partner collaboration over the Internet. Just as the World Wide Web began as a technology for scientific collaboration and was adopted for e-business, it is expected a similar trajectory for Grid technologies.

Database grid has been an important direction in Grid Computing that adds lots of features and new challenges in many different areas: Distributed Data Replication in Grid Computing, Grid Database Services, Grid Database Access and Integration, Security in Database Grid, etc...

For this a high-performance distributed data manager should be developed that allows very efficient execution of database queries involving numerical and other data. Even though very high performance is attained by utilizing many main-memory database engines running on PCs and connected through the Grid [13] but this is not enough. A new protocol for distributed transaction should be formed, that can be suitable for parallel and distributed execution in a Grid environment. This protocol should aim at achieving higher concurrency of multiple transactions. It will be also very suitable for concurrent execution of long transactions.

In this thesis we have tried first to identify the specifications that the algorithm has to take into consideration while the key transaction processing features of a DBMS should be maintained which are the ACID properties that stand for Atomicity, Consistency, Isolation and Durability. These characteristics are mainly the size of the data, no deadlocks or abort due to conflicts should be generated, since the host nodes are by large stateless and require very little persistency, if a process fails its active transactions (which are un-committed by definition) are cancelled and eventually restarted so we have to take into account the transient hosts in a grid computing environment, we should have minimal reads and writes to disk storage. Calculations are likely to be satisfied by cached records and the algorithm must be suitable for distributed execution.

And then we tried to incorporate, in this algorithm, techniques which are suitable for execution in a grid of transient hosts.

This new algorithm is mainly based on the two existing concurrency control algorithms which are Timestamp and Optimistic algorithms. In this thesis we tried to solve the common issue of abort and restart the entire transaction in case of conflicts between the concurrent transactions in order to save time that is to improve performance and to achieve a higher concurrency level.
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIGURE 1</td>
<td>DISTRIBUTED TRANSACTION</td>
<td>24</td>
</tr>
<tr>
<td>FIGURE 2</td>
<td>COMMUNICATION STRUCTURE OF CENTRALIZED 2PL</td>
<td>29</td>
</tr>
<tr>
<td>FIGURE 3</td>
<td>COMMUNICATION STRUCTURE OF DISTRIBUTED 2PL</td>
<td>30</td>
</tr>
<tr>
<td>FIGURE 4</td>
<td>PHASES OF PESSIMISTIC TRANSACTION EXECUTION</td>
<td>35</td>
</tr>
<tr>
<td>FIGURE 5</td>
<td>PHASE OF OPTIMISTIC TRANSACTION EXECUTION</td>
<td>35</td>
</tr>
<tr>
<td>FIGURE 6</td>
<td>CASCADING COMMIT OF CONCURRENT TRANSACTIONS</td>
<td>40</td>
</tr>
<tr>
<td>FIGURE 7</td>
<td>SERIAL EXECUTION</td>
<td>42</td>
</tr>
<tr>
<td>FIGURE 8</td>
<td>CONCURRENT EXECUTION SCENARIO</td>
<td>43</td>
</tr>
<tr>
<td>FIGURE 9</td>
<td>TIMESTAMP ALGORITHM</td>
<td>46</td>
</tr>
<tr>
<td>FIGURE 10</td>
<td>OPTIMISTIC ALGORITHM</td>
<td>47</td>
</tr>
<tr>
<td>FIGURE 11</td>
<td>PROPOSED ALGORITHM</td>
<td>47</td>
</tr>
</tbody>
</table>
List of Acronyms

ACID: Atomicity, Consistency, Isolation, Duration.
API: Application Programming interface
CORBA: Common Object Request Broker Architecture
DCOM: Distributed Component Object Model
GASS: Global Access to Secondary Storage
GCM: Global Communications Manager
GDM: Grid Data Manager
GIS: Grid Information Service
GRAM: Globus Resource Allocation manager
GSI: Grid Security Infrastructure
GTM: Global Transaction Manager
HBM: Heartbeat Monitor
IDL: Interface Definition Language
LDAP: Lightweight Directory Access Protocol
LM: Lock Manager
MDS: Monitoring and Discovery Service
MPI: Message Passing Interface
OGSA: Open Grid Service Architecture
RAC: Real Application Clusters
RL: Read Lock
ROWA: Read Once Write After
RSL: Resource Specification Language
SC: Scheduler
TM: Transaction Manager
VO: Virtual Organization
WAL: Write-Ahead-Log
WL: Write Lock
Table of Contents

CHAPTER 1: PROBLEM DEFINITION ............................................................ 1
  1.1 Introduction ................................................................................. 2
  1.2 Problem Definition ................................................................. 2
  1.3 Thesis Organization ................................................................. 3

CHAPTER 2: GRID COMPUTING ............................................................... 4
  2.1 Grid Computing ........................................................................ 5
  2.2 What is Enterprise Grid Computing? ........................................... 6
  2.3 GRID ARCHITECTURE .............................................................. 7
  2.4 WHAT IS GLOBUS? ................................................................. 8
  2.5 WHAT IS THE GLOBUS TOOLKIT? ............................................. 8
  2.6 GLOBUS ARCHITECTURE ....................................................... 9
    Layer 1: Grid Fabric ................................................................. 9
    Layer 2: Grid Services ............................................................. 9
      GRAM .................................................................................... 9
      GIS ..................................................................................... 9
      GSI ................................................................................... 10
    Layer 3: Application Toolkits ................................................... 10
    Layer 4: Specific Applications ................................................ 10
  2.7 How hard it is to use the Grid in an application: ......................... 10
  2.8 Does Globus solve all the problems? ........................................ 11
  2.9 Don't Java and JINI solve all these problems? ......................... 11
  2.10 Doesn't CORBA solve all these problems? ............................... 11
  2.11 Doesn't DCOM solve all these problems? ............................... 11
  2.12 Other Perspectives on Grids .................................................. 12

CHAPTER 3: GRID DATABASE ............................................................... 14
  3.1 Grid Data System .................................................................... 15
    3.1.1 The Need for Databases in Grid Environments .................... 16
  3.2 High-performance GRID Database Manager: ............................. 16
    3.2.1 Grid Enterprise Applications ........................................... 17
    3.2.2 Grid Design ................................................................. 17
  3.3 Oracle Technologies that Enable the Grid .................................. 18
  3.4 Oracle 10g concurrency and consistency .................................... 19
    3.4.1 Concurrency ............................................................... 19
    3.4.2 Read Consistency ........................................................ 20
    3.4.3 Read Consistency, Undo Records, and Transactions ............ 20
    3.4.4 Locking Mechanisms ................................................... 21
    3.4.5 Automatic Locking ....................................................... 21

CHAPTER 4: DISTRIBUTED TRANSACTIONS .......................................... 22
  4.1 ACID: .................................................................................... 23
  4.2 Distributed Transaction: .......................................................... 24
  4.3 Role of Distributed Execution Monitor ....................................... 25
  4.4 Serializability Theory: ............................................................. 25
    4.4.1 Distributed Serializability .............................................. 26
4.5 CONCURRENCY CONTROL TECHNIQUES .................................................. 27
4.5.1 LOCKING-BASED CONCURRENCY CONTROL ALGORITHMS .............. 28
  4.5.1.1 Centralized 2 PL: ................................................................. 29
  4.5.1.2 Primary Copy 2PL: ............................................................ 30
  4.5.1.3 Distributed 2PL: ............................................................... 30
4.5.2 TIMESTAMP-BASED CONCURRENCY CONTROL ALGORITHMS .......... 31
  4.5.2.1 Basic TO Algorithm: .......................................................... 32
  4.5.2.2 Conservative TO Algorithm: ............................................. 33
  4.5.2.3 Multiversion TO Algorithm .............................................. 34
4.5.3 OPTIMISTIC CONCURRENCY CONTROL ALGORITHMS ....................... 35

CHAPTER 5:  ......................................................................................... 37
PROPOSED CONCURRENT TRANSACTION ALGORITHM ................................ 37
  5.1 Proposed Algorithm: ................................................................. 38
  5.2 Scenario of Execution ............................................................... 41
    5.2.1 Serial Execution Scenario .................................................. 42
    5.2.2 Concurrent Execution Scenario ........................................... 43
  5.3 Comparison between the timestamp based algorithms and the proposed
    algorithm .................................................................................. 44
  5.4 Comparison between the Optimistic based algorithm and the proposed
    algorithm: .................................................................................. 45
  5.5 Summary Tables ........................................................................ 46
    5.5.1 Timestamp Algorithm: ......................................................... 46
    5.5.2 Optimistic Algorithm: ......................................................... 47
    5.5.3 Proposed Algorithm: ........................................................... 47
    5.5.4 Further Examples and Testing: ............................................. 48

CHAPTER 6:  ......................................................................................... 52
CONCLUSION AND FUTURE RESEARCH .............................................. 52

REFERENCES ..................................................................................... 54

-vii-
CHAPTER 1:

PROBLEM DEFINITION
1.1 Introduction

From high speed VPNs to server clusters to Web services integration platforms -- THE FUTURE IS NOW GRID TECHNOLOGY.

In much the same way the Internet began as an effort to share computing resources among academic researchers before spreading to businesses and then to consumers, GRID computing will be used by enterprises. More important, research into GRID computing will lead to the creation of software that will solve critical problems that companies now face in building distributed inter-enterprise applications.

The Grid was originally designed as a large network of computer systems able to offer an environment where computing and storage resources are shared on demand.

To date, the development of standards, such as the Open Grid Service Architecture (OGSA), along with the introduction of new paradigms, such as the Semantic Grid [10], is leading the Grid toward an environment that is not only suited for computational-intensive applications, but also for computing scenarios typical for distributed systems, like service and information providing, multimedia environments, ubiquitous computing, etc.

For these and other reasons, the Grid is becoming an interesting and challenging environment supporting both old and new services for cooperative applications. A relevant research effort is thus needed not only to investigate innovative Grid infrastructure but also to make the current Grid model suitable for these emerging usage scenarios.

Current research is now being done on developing a new kind of database manager utilizing the evolving Grid infrastructure for distributed computation, called the GRID Data Manager (GDM) that should be having very high performance and support for customizable representation of streamed data in distributed data and computation servers [1,21].

1.2 Problem Definition

With distributed geographical databases comes the problem of concurrency control, i.e., ensuring that database operations from different users do not interfere with each other. For example, queries issued from the GIS application domain often access a large portion of the database, perform lengthy operations, pauses for input from users or a combination of above; transactions may last for hours, days, or even months.

Such properties lead to special requirement specifications that a transactional data server has to meet.

The objective of my thesis is to make a proposal regarding a challenging problem that still need to be solved which is to develop a protocol of concurrency control for multiple concurrent transactions while keeping the consistency and integrity of the grid database. A new design for a Grid DBMS, which should provide highly scalable
and highly available services. We based ourselves on existing algorithms that we
enhanced to make them suitable for grid database, these algorithms are Timestamp
and Optimistic that were both extended to fulfill the wished result.

This protocol is suitable for distributed transaction and for parallel and distributed
execution in a Grid environment. It will also be suitable for execution of long
transactions due to the fact that no deadlocks or aborts will be generated due to
conflicts. This protocol should aim at achieving higher concurrency control and it
should ensure that transactions are executed safely and that they follow the ACID
rules.

1.3 Thesis Organization

The thesis is composed of six chapters with the aim of presenting the grid new
technology and a solution for the concurrency control problem that is still need to be
solved in a grid database environment.

The second chapter is dedicated to the entire Grid Computing technology, its
benefits, its architecture and the tools developed so far for the Grid infrastructure.
Chapter 3 talks more about databases in the grid and the need for high performance
grid data manager in order to be able to support the enterprise grid applications.
Chapter 4 explains in details all the concurrency control algorithms used in the
distributed database environment. Chapter 5 presents the proposed concurrent
transaction algorithm into a grid database and a comparison between the proposed
algorithm and the concurrent transactions algorithms which are Timestamp and
Optimistic algorithms. Finally we have chapter 6 for the conclusion and the future
research.
CHAPTER 2:
GRID COMPUTING
The Computational Grid represents a rapidly emerging and expanding technology that allows geographically distributed resources (CPU cycles, data storage, sensors, visualization devices, and a wide variety of Internet-ready instruments), which are under distinct control, to be linked together in a transparent fashion [15]. The power of the Grid lies not only in the aggregate computing power, data storage, and network bandwidth that can readily be brought to bear on a particular problem, but on its ease of use.

Grids are now a viable solution to certain computationally- and data-intensive computing problems for the following reasons: (a) The Internet is reasonably mature and able to serve as fundamental infrastructure. (b) Network bandwidth has increased to the point of being able to provide efficient and reliable services. (c) Storage capacity has now reached commodity levels, where one can purchase a terabyte of disk for roughly the same price as a high-end PC. (d) Many instruments are Internet-aware. (e) Clusters, supercomputers, storage and visualization devices are becoming more easily accessible. (f) Applications have been parallelized. (g) Collaborative environments are moving out of the alpha phase of implementation.

For these and other reasons, grids are starting to move out of the research laboratory and into early-adopter production systems. The focus of grid deployment continues to be on the difficult issue of developing high quality middleware. Many types of computational tasks are naturally suited to grid environments, including data-intensive applications.

### 2.1 Grid Computing

Grid computing refers to applying the power and resources of many computers in a network to address computing problems that require a great number of computer processing cycles and the ability to manage extremely large amounts of data. Grid computing links many different computers from several locations into one large virtual computer [1], making all of the collective computing power available to the users in the enterprise.

The central idea of grid computing is that computing should be as reliable, pervasive, and transparent as a utility. It shouldn’t matter where the data or application resides, or what computer processes the user’s request. Information or computation should be requested and should be delivered. [8].

By doing this, it turns processing power into a utility, like water or electricity that can be bought and used as needed, depending on the job at hand.

Grids are super Internets for high-throughput and high-performance computing: worldwide collections of resources such as computer clusters, supercomputers, data storage and advanced instruments. These resources and their users are often separated by great distances and connected by high-speed networks.

Grid Computing is where there is a network of computers which tap into a main server. The concept comes from the electrical grid and would be arranged in a
system that functions in a similar fashion. If a client takes an appliance and plugs it into a wall outlet, then he becomes a client of the electrical grid. He doesn’t know how the grid is implemented, whether the power station is in the next state or next door. All he wants is power; he plugs in and he gets it. That’s the highest logical level of Grid Computing [6].

Today, islands of computing within organizations make inefficient use of resources. Systems are slow to change and expensive to maintain. Grid computing addresses these problems by providing an adaptive software infrastructure that makes efficient use of low-cost servers and modular storage, which balances workloads more effectively and provides capacity on demand. By scaling out with small servers in small increments, performance and reliability at low-cost is assured. New unified management allows managing everything cheaply and simply in the grid.

Grid computing removes the fixed connections between applications, servers, databases, machines, storage – every component of the grid. By treating everything in the grid as a virtualized service, intelligent systems can optimize resource utilization and responsiveness.

Grid computing is based on five fundamental attributes: virtualization, dynamic provisioning, resource pooling, self-adaptive systems, and unified management.

2.2 What is Enterprise Grid Computing?

Enterprise grid computing builds a critical software infrastructure that can run on large numbers of small, networked computers, by combining two related concepts:

- **Implement One from Many.** Grid computing coordinates the use of clusters of machines to create a single logical entity, such as a database or an application server. By distributing work across many servers, grid computing exhibits benefits of availability, scalability, and performance using low-cost components. Because a single logical entity is implemented across many machines, companies can add or remove capacity in small increments, online. With the capability to add capacity on demand to a particular function, companies get more flexibility for adapting to peak loads, thus achieving better hardware utilization and better business responsiveness.

- **Manage Many as One.** Grid computing allows managing and administering groups of machines, groups of database instances, and groups of application servers at low-cost. Grid computing first removes many of the administrative costs of managing a single system by making each database and each application server adaptive to changing circumstances. Then, the model makes managing many systems simple, by allowing them to be managed as a single logical entity.
Primary benefit of grid computing to businesses is achieving high quality of service and flexibility at lower cost. Enterprise grid computing lowers costs by:

1. Increasing hardware utilization and resource sharing
2. Enabling companies to scale out incrementally with low-cost components
3. Reducing management and administration requirements

2.3 GRID ARCHITECTURE

Grids are the amalgamation of entities called virtual organizations. A virtual organization contains individuals or institutions that provide a resource which contributes to the grid community.

Grids, irrespective of type, have the following components:

- Underlying Fabric: The underlying fabric represents infrastructure which performs switching and routing. It provides the connectivity between disparate resources. It is the network infrastructure. It should provide and meet the requirements of High Performance and Distributed Computing.
- Computational Elements (Infrastructure): Infrastructure represents the nodes in the system which perform real work. These nodes may be computational elements such as clusters, storage resources or even instrumentation.
- Middleware technologies: Middle-ware technologies include schedulers, authentication regimes, management tools, API's and other interfaces which make the underlying infrastructure usable. The middleware hides the complexities of grid computing from the users; it devises protocols and services which leverage the services of grids. It consists of programming models, schedulers and brokers to distribute jobs in an orderly fashion and it provides management, debugging and profiling tools to a user.
- Applications: A number of applications exist that leverage this technology – applications tend to fall into the following classifications:
  - Experimentation
  - Simulation
  - Theoretical – (modeling etc)
2.4 WHAT IS GLOBUS?

The Globus Project is a research and development project focused on enabling the application of grid concepts to scientific and engineering computing.

- Groups around the world are using the Globus Toolkit to build Grids and to develop Grid applications.
- Globus research targets technical challenges that arise from these activities. Typical research areas include resource management, data management and access, application development environments, information services, and security.
- Globus software development has resulted in the Globus Toolkit, a set of services and software libraries to support Grids and Grid applications. The Toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection, and portability.

The Globus software toolkit facilitates the creation of usable Grids, enabling high-speed coupling of people, computers, databases, and instruments. With Globus, jobs can run on two or more machines at the same time, even though the machines might be located far apart and owned by different organizations. Globus software helps scientists deal with very large datasets and complex remote collaborations.

Globus is an infrastructure toolkit that provides middleware for communication, resource location and allocation, security, information, and data access services. Each service has a defined interface and implementation notes [20].

2.5 WHAT IS THE GLOBUS TOOLKIT?

The Toolkit is first and foremost a "bag of services," a set of useful components that can be used either independently or together to develop useful grid applications and programming tools [12].

- The Globus Resource Allocation Manager (GRAM) provides resource allocation and process creation, monitoring, and management services. GRAM implementations map requests expressed in a Resource Specification Language (RSL) into commands to local schedulers and computers.
- The Grid Security Infrastructure (GSI) provides a single-sign-on, run-anywhere authentication service, with support for local control over access rights and mapping from global to local user identities. Smartcard support increases credential security.
- The Monitoring and Discovery Service (MDS) is an extensible Grid information service that combines data discovery mechanisms with the Lightweight Directory Access Protocol (LDAP). MDS provides a uniform framework for providing and accessing system configuration and status information such as compute server configuration, network status, or the locations of replicated datasets.
- Global Access to Secondary Storage (GASS) implements a variety of automatic and programmer-managed data movement and data access strategies, enabling programs running at remote locations to read and write local data.
Concurrent Transaction Execution in Grid Database

- Nexus and globus_io provide communication services for heterogeneous environments, supporting multimethod communication, multithreading, and single-sided operations.
- The Heartbeat Monitor (HBM) allows system administrators or ordinary users to detect failure of system components or application processes.

For each component, an application programmer interface (API) written in the C programming language is provided for use by software developers. Command line tools are also provided for most components, and Java classes are provided for the most important ones. Some APIs make use of Globus servers running on computing resources.

A large number of individuals, organizations, and projects have developed higher-level services, application frameworks, and scientific/engineering applications using the Globus Toolkit. For example, the Condor-G software provides an excellent framework for high-throughput computing (e.g., parameter studies) using the Globus Toolkit for inter-organizational resource management, data movement, and security [4, 20].

2.6 GLOBUS ARCHITECTURE

The Globus architecture follows the "Layered bag of services" model. A typical application will only use a subset of services and build some custom tailored tools on top of them [20].

Layer 1: Grid Fabric

The fabric of the Grid comprises the underlying systems, computers, operating systems, networks, storage systems, and routers - the building blocks.

Layer 2: Grid Services

Grid services integrate the components of the Grid fabric. Examples of the services provided by Globus are

GRAM

The Globus Resource Allocation Manager, GRAM, is a basic library service that provides capabilities to do remote-submission of job start up. GRAM unites Grid machines, providing a common user interface so that a job can be submitted to multiple machines on the Grid fabric. GRAM is a general, ubiquitous service, with specific applications toolkits commands built on top of it.

GIS

The Grid Information Service, GIS, formerly known as the Metacomputing Directory Service, MDS, provides information service. GIS is queried to discover the properties of the machines, computers and networks that are to be used: how many processors are available at the moment? What bandwidth is provided? Is the storage on tape or disk? Where are the replicas of a given file? Using an LDAP (Lightweight Directory
Access Protocol) Server, GIS provides middleware information in a common interface to put a unifying picture on top of a disparate equipment.

**GSI**

The Grid Security Infrastructure, GSI, is a library for providing generic security services for applications that will be run on the Grid. Application programmers use the gss-api library or command line tools for adding authentication to a program. GSI provides programs, such as grid-proxy-init, to facilitate login to a variety of sites, while each site has its own security measures. That is, on the fabric layer, the various machines the user wants to use might be governed by disparate security policies; GSI provides a means of simplifying multiple remote logins.

**Layer 3: Application Toolkits**

Application toolkits use Grid Services to provide higher-level capabilities, often targeted to specific classes of applications. For example, the Globus development team has created a set of Grid service tools and a toolkit of programs for running remotely distributed jobs. These include remote job submission commands (globusrun, globus-job-submit, globus-job-run), built on top of the GRAM service, and MPICH-G, a Grid-enabled implementation of the Message Passing Interface (MPI).

A number of groups are also developing a range of other toolkits such as support for distributed management of large datasets, collaborative visualization, and online instrumentation.

**Layer 4: Specific Applications**

A variety of applications can and have been developed that build on services provided by the three layers just described. Such applications include high-energy physics, cosmology, chemical engineering, climate, combustion and astrophysics.

**2.7 How hard it is to use the Grid in an application:**

It might not be hard. Some applications can run with no modification at all, simply by linking with a Grid-enabled version of an appropriate programming library.

Even when appropriate high-level tools are not available, an important feature of the Globus toolkit is that Globus capabilities can often be incorporated into an existing application incrementally, producing a series of increasingly "grid-enabled" versions. For example, a grid-enabled distributed interactive simulation application, SF-Express, was developed as follows:

- They used resource management services to simplify program startup on multiple computers, avoiding separate logins and scheduler commands to each system.
- They then introduced a Globus library called DUROC to coordinate program startup across multiple sites, addressing certain fault detection and recovery strategies.
Concurrent Transaction Execution in Grid Database

• They then used Globus data access services to stage executables and configuration files to the remote systems, and to direct output streams to the originating location.

2.8 Does Globus solve all the problems?

Certainly not! The Globus services that have been developed to date help users overcome some of the barriers to Grid computing. However, many challenging problems remain to be overcome before we can say that we have a fully functional Grid. Among the major research challenges that others and we are addressing in current work are:

• End-to-end resource management and adaptation techniques able to provide application-level performance guarantees despite dynamic resource properties.
• Automated techniques for negotiation of resource usage, policy, and accounting in large-scale grid environments.
• High-performance communication methods and protocols.
• Infrastructure and tool support for data-intensive applications, advance tele-immersion concepts, and new problem solving environment techniques.

2.9 Don't Java and JINI solve all these problems?

Java provides useful technology for portable, object-oriented application development, but it does not address many of the hard problems that arise when we try to achieve high-performance execution in heterogeneous distributed environments. For example, Java doesn't help us run programs on different types of supercomputers, discover the policy elements that apply at a particular site, achieve single sign-on authentication, perform high-speed transfer across wide area networks, etc. The Globus toolkit addresses some of these concerns, and uses Java to provide portable clients.

2.10 Doesn't CORBA solve all these problems?

The Common Object Request Broker Architecture (CORBA) defines a standard Interface Definition Language (IDL) for inter-language interoperability, a remote procedure call service, and a variety of more specialized services such as a trader (for resource location). Like Java, CORBA provides important software engineering advantages, but it doesn't directly address the challenges that arise in Grid environments such as specialized devices and the high performance required by many Grid applications. Again, Grid and CORBA technologies are complementary, not competing.

2.11 Doesn't DCOM solve all these problems?

Microsoft's Distributed Component Object Model (DCOM) provides a variety of services, including remote procedure call, directory service, and distributed file system. Again, these services are useful but don't address issues of heterogeneity or performance directly.
2.12 Other Perspectives on Grids

The perspective on Grids and VOs is of course not the only view that can be taken. A summary is found here—and critique—some alternative perspectives (given in italics).

*The Grid is a next-generation Internet.* "The Grid" is not an alternative to "the Internet": it is rather a set of additional protocols and services that build on Internet protocols and services to support the creation and use of computation- and data-enriched environments. Any resource that is "on the Grid" is also, by definition, "on the Net."

*The Grid is a source of free cycles.* Grid computing does not imply unrestricted access to resources. Grid computing is about controlled sharing. Resource owners will typically want to enforce policies that constrain access according to group membership, ability to pay, and so forth. Hence, accounting is important, and a Grid architecture must incorporate resource and collective protocols for exchanging usage and cost information, as well as for exploiting this information when deciding whether to enable sharing.

*The Grid requires a distributed operating system.* In this view, Grid software should define the operating system services to be installed on every participating system, with these services providing for the Grid what an operating system provides for a single computer: namely, transparency with respect to location, naming, security, and so forth. Put another way, this perspective views the role of Grid software as defining a virtual machine. However, we feel that this perspective is inconsistent with our primary goals of broad deployment and interoperability. We argue that the appropriate model is rather the Internet Protocol suite, which provides largely orthogonal services that address the unique concerns that arise in networked environments. The tremendous physical and administrative heterogeneities encountered in Grid environments means that the traditional transparencies are unobtainable; on the other hand, it does appear feasible to obtain agreement on standard protocols. The architecture proposed here is deliberately open rather than prescriptive: it defines a compact and minimal set of protocols that a resource must speak to be “on the Grid”; beyond this, it seeks only to provide a framework within which many behaviors can be specified.

*The Grid requires new programming models.* Programming in Grid environments introduces challenges that are not encountered in sequential (or parallel) computers, such as multiple administrative domains, new failure modes, and large variations in performance. However, we argue that these are incidental, not central, issues and that the basic programming problem is not fundamentally different. As in other contexts, abstraction and encapsulation can reduce complexity and improve reliability. But, as in other contexts, it is desirable to allow a wide variety of higher-level abstractions to be constructed, rather than enforcing a particular approach. So, for example, a developer who believes that a universal distributed shared memory model can simplify Grid application development should implement this model in terms of Grid protocols, extending or replacing those protocols only if they prove inadequate for this purpose. Similarly, a developer who believes that all Grid resources should be presented to users as objects needs simply to implement an object-oriented "API" in terms of Grid protocols.

*The Grid makes high-performance computers superfluous.* The hundreds, thousands, or even millions of processors that may be accessible within a VO represent a significant source of computational power, if they can be harnessed in a useful fashion. This does not imply, however, that traditional high-performance computers are obsolete. Many problems require tightly coupled computers, with low latencies...
and high communication bandwidths; Grid computing may well increase, rather than reduce, demand for such systems by making access easier.
CHAPTER 3:
GRID DATABASE
3.1 Grid Data System

Initially, Grid technologies were principally associated with supercomputer centers and large-scale scientific applications in physics and astronomy [9]. They are now increasingly seen as being relevant to many areas of e-Science and e-Business. The emergence of the Open Grid Services Architecture (OGSA), to complement the ongoing activity on web services standards, promises to provide a service-based platform that can meet the needs of both business and scientific applications. Early Grid applications focused principally on the storage, replication and movement of file-based data. Now the need for the full integration of database technologies with Grid middleware is widely recognized. Not only do many Grid applications already use databases for managing metadata, but also increasingly many are associated with large databases of domain-specific information (e.g. biological or astronomical data).

Grid technologies enable sharing of data from different sites by creating a virtual organization of the data. Therefore, the grid is able to help reduce the single point of failure inherited in a centralized database system. New research results can be stored on a local system and shared with the research community immediately. Users no longer need to know the location of their target information, but are able to access and retrieve in a transparent manner.

Having different sources of data in different locations raises compatibility issues in the design of the grid computing infrastructure [14]. The middleware layer therefore will interface with the grid services, such as security infrastructure and allocation manager, to transform the data from different sites to a standard format. In addition to providing a seamless access to the data repository, the grid infrastructure is ideal for parallel/distributed applications [18]. Since the grid is a collection of computer-related resources, geographically distributed computing power can also be utilized in a similar manner to the aforementioned data-sharing concept. Computationally intensive tasks such as dynamics simulation or network modeling can be executed efficiently using all the computational resources available on the grid. The primary challenge is to create a software system that can manage the distributed computing components so that they can access distributed data efficiently, i.e., minimizing the incurred communication costs by deciding if data are going to be sent to a system for computing or if a computation object (i.e., program) should actually be executed where the data are located.

The location(s) of the requested data will be supplied by the application's search engine running on the underlying grid data system. Once the locations are identified, data that allow the fastest access will be selected and processed according to the specified computation method. This process can be recursively applied in cases where the computation requires more input that cannot be found locally.

The grid security system must be implemented as a gateway for every grid site providing uniform access to heterogeneous storage systems, which can be databases from different vendors.

Furthermore, the grid security system will support the single signed-on capability so that the access authentication should be verified merely the first time. The storage system application programming interface (API) will provide the uniform accessing service to different (heterogeneous) storage systems. Users need not know how each storage system is operated.
3.1.1 The Need for Databases in Grid Environments

In many scientific and commercial domains, database management systems have a central role in data storage, access, organization, and authorization for numerous applications. Early Grid applications, mainly motivated by an attempt to cater to scientific-based computing, were often closely associated with devices or tools that read and/or generated flat files. Consequently, support for files rather than for the management of structured data had the highest profile in the early Grid toolkits. However, over time, the file management systems and registries associated with Grid toolkits themselves became complex, and database management systems were increasingly used to store Grid metadata. Contemporaneously, the requirements of the scientific computing community have become more sophisticated with, for example, biological and astronomical communities generating large quantities of data that increasingly use databases for storage and retrieval. Similarly, engineering, medical research, healthcare and many governmental systems can also take advantage of Grids that access and integrate multiple and distributed collections of structured data. This data needs to be made available and accessible to distributed sets of users and their applications, which makes these communities ideal candidates to adopt Grid-based infrastructures [13].

3.2 High-performance GRID Database Manager:

The GRID initiative will allow for exchanging and processing very large amounts of data. There are projects which are geographically distributed and utilizing very high performance grid computer. As the volume of data processed is very large and dynamic there will be need for very high performing data management systems. For this a high-performance stream oriented distributed data manager is being developed that allows very efficient execution of database queries to streamed data involving numerical and other data. Very high performance is attained by utilizing many object-relational main-memory database engines running on PCs and connected through the GRID. [5, 9]

Of particular interest is the development of new distributed data population and query processing techniques for this kind of applications and thereby utilizing distributed and scalable data structures for high-performance stream data processing.

Current research is now being done on developing a new kind of database manager utilizing the evolving Grid infrastructure for distributed computation, called the GRID Data Manager (GDM) that should be having very high performance and support for customizable representation of streamed data in distributed data and computation servers. The target application area is space physics, atmospheric physics, etc... [21]

Just as the World Wide Web began as a technology for scientific collaboration and was adopted for e-business, it is expected a similar trajectory for Grid technologies. It is expected that similar applications to become important in commercial settings, initially for scientific and technical computing applications and then for commercial distributed computing applications, including enterprise application integration and business to business partner collaboration.
This is a new environment very different from the conventional server-oriented database environments. A research challenge is to make GDM able to handle very large amounts of dynamically produced distributed data. All this will be achieved by utilizing cheap and large main-memories (on PCs) connected through the GRID to form clusters of main-memory. The main memory of modern computers can be cheap and large enough to entirely store many databases.

GRID-based clusters of main memory database nodes combined with extensibility through user-defined data representations provide very high performance [5]. The databases are scalable by dynamically incorporating new GRID nodes as the database grows.

Efficient customized data representations and query optimization algorithms can thus be distributed to each node in the cluster.

3.2.1 Grid Enterprise Applications

Grid technologies enable sharing of data from different sites by creating a virtual organization of the data. The current grid-enabling software technology, allows the sharing of geographically distributed data. Therefore, the grid is able to help reduce the single point of failure inherited in a centralized database system.

Having different sources of data in different locations raises compatibility issues in the design of the grid computing infrastructure.

Many multi-tier component-based architectures (e.g. .NET, J2EE) are ultimately bottle-necked at the database tier. Any type of grid requires a core of highly-reliable directory service, which ideally would be implemented on the grid itself.

Other Grid DB designs have innovated in functionality or semantics. The protocol that should be designed should focus on concurrency and availability.

3.2.2 Grid Design

The queries and transactions are assigned to processes which execute on grid nodes. The design is meant to identify proven techniques which are suitable for execution in a grid of transient hosts.

Worker Processes: handle queries from multiple clients. Worker Processes interact with the storage Processes to read and write records, and interact among themselves to synchronize transactions.

Storage Processes: These control directly-attached or shared storage. They interact among themselves to provide storage which is persistent, redundant and fault tolerant.

Grid Nodes: Grid nodes are the actual hosts on the grid executing processes. Storage nodes (i.e. executing a storage process) are expected to show some degree of stability. Failure of such nodes leads to some performance degradation.
Worker nodes are stateless by design and may exhibit transient behavior (fail, restart) with little effect on system performance.

Transaction-Execution: A transaction execution is arbitrarily assigned to a worker process which determines the required tables and keys. The transaction is time stamped a shared/global counter. Execution proceeds to the concurrency algorithm.

Distributed-Caching: The main memory of the worker- and storage- nodes from a memory-based distributed cache. This can increase the read performance of the aggregate storage.

Write-ahead-log: Several nodes are designated as a fast and redundant write-ahead-log. The WAL is required to transparently recover from a failure by a worker process during the final write-commit stage of a transaction. Subtle synchronization is required when records are in the WAL.

Consider a case where the updates to the database are written into the stable storage before the log is modified in stable storage to reflect the update. If a failure occurs before the log is written the database will remain in updated form, but the log will not indicate the update that makes it impossible to recover the database to a consistent and up-to-date state. Therefore, the stable log is always updated prior to the updating of the stable database. This is know as the write-ahead-logging (WAL) protocol and it can be precisely specified as follows:

1- Before a stable database is updated (perhaps due to actions of a yet uncommitted transaction), the before images should be stored in the stable log. This facilitates undo.

2- When a transaction commits, the after images have to be stored in the stable log prior to the updating of the stable database. This facilitates redo.

Storage-Redundancy: All data must be duplicated. Secondary copies may be suitable for data dispersal (distribution) schemes. Such schemes impose a high performance overhead but may provide continued availability during multiple failures.

3.3 Oracle Technologies that Enable the Grid

Oracle has been working for years on technologies that support and enable grid computing. Computing resource provisioning is one of the most important capabilities of a grid. This enables computing resources to be dynamically provisioned to applications as required. Resources must be appropriately allocated based on business priorities and demand. Oracle provides a number of features for computing resource provisioning, including: [7]

Real Application Clusters: RAC is a cluster database with a shared cache architecture that runs on multiple machines, attached through a cluster interconnect and a shared storage subsystem. An Oracle RAC database not only appears like a single standard Oracle Database to users, but the same maintenance tools and practices used for a single Oracle Database can be used on the entire cluster. All standard backup and recovery operations, including the use of Recovery Manager, work transparently with RAC. All SQL operations, including data definition language
and integrity constraints, are also identical for both configurations. The most important part of RAC, however, is the ability to manage the workload—to add nodes or relinquish nodes on demand—based on the business processing needs. [7]

**Automatic Storage Management:** Oracle recommends using Automatic Storage Management (ASM) for the database files and a cluster file system for the Oracle home. ASM simplifies the administration of Oracle database files. Instead of managing many database files, ASM requires managing only a small number of disk groups. A particular disk group can be defined as the default disk group for a database.

**Oracle Resource Manager:** Though Oracle Database is largely a self-managing database, Database Resource Manager allows resource administrators to influence how the Oracle database resources are allocated to users.

**Oracle Scheduler:** Oracle Scheduler provides many capabilities to schedule and perform business and IT tasks, called jobs, in a grid.

Information provisioning means delivering information to users whenever they need it, regardless of where it resides on the grid. To process information on any available resource, the grid must efficiently share information across distributed systems. The grid must also provide access to data residing on heterogeneous systems—database systems from multiple vendors and file systems. Oracle provides a broad set of features and tools for information provisioning on a grid, including the following:

**Oracle Transportable Tablespaces:** Transportable Tablespaces allows Oracle data files to be unplugged from a database, moved, or copied to another location, and then plugged into another database. Unplugging or plugging a data file involves reading or loading only a small amount of metadata. Transportable Tablespaces also supports simultaneous mounting of read-only table spaces by two or more databases.

**Oracle Streams:** Some data needs to be shared as it is created or changed, rather than occasionally shared in bulk. Oracle Streams can stream data between databases, nodes, or blade farms in a grid and can keep two or more copies in sync as updates are applied. It also provides a unified framework for information sharing, combining message queuing, replication, events, data warehouse loading, notifications, and publish/subscribe into a single technology. [7, 11]

### 3.4 Oracle 10g concurrency and consistency

#### 3.4.1 Concurrency

A primary concern of a multi-user database management system is how to control concurrency, which is the simultaneous access of the same data by many users. Without adequate concurrency controls, data could be updated or changed improperly, compromising data integrity.
One way to manage data concurrency is to make each user wait for a turn. The goal of a database management system is to reduce that wait so it is either nonexistent or negligible to each user. All data manipulation language statements should proceed with as little interference as possible, and destructive interactions between concurrent transactions must be prevented. Destructive interaction is any interaction that incorrectly updates data or incorrectly alters underlying data structures. Neither performance nor data integrity can be sacrificed.

Oracle resolves such issues by using various types of locks and a multiversion consistency model. These features are based on the concept of a transaction. It is the application designer's responsibility to ensure that transactions fully exploit these concurrency and consistency features. [11]

### 3.4.2 Read Consistency

Read consistency, as supported by Oracle, does the following:

- Guarantees that the set of data seen by a statement is consistent with respect to a single point in time and does not change during statement execution (statement-level read consistency)
- Ensures that readers of database data do not wait for writers or other readers of the same data
- Ensures that writers of database data do not wait for readers of the same data
- Ensures that writers only wait for other writers if they attempt to update identical rows in concurrent transactions

The simplest way to think of Oracle's implementation of read consistency is to imagine each user operating a private copy of the database, hence the multiversion consistency model. [11]

### 3.4.3 Read Consistency, Undo Records, and Transactions

To manage the multiversion consistency model, Oracle must create a read-consistent set of data when a table is queried (read) and simultaneously updated (written). When an update occurs, the original data values changed by the update are recorded in the database undo records. As long as this update remains part of an uncommitted transaction, any user that later queries the modified data views the original data values. Oracle uses current information in the system global area and information in the undo records to construct a read-consistent view of a table's data for a query.

Only when a transaction is committed are the changes of the transaction made permanent. Statements that start *after* the user's transaction is committed only see the changes made by the committed transaction.

The transaction is key to Oracle's strategy for providing read consistency. This unit of committed (or uncommitted) SQL statements:

- Dictates the start point for read-consistent views generated on behalf of readers
Concurrent Transaction Execution in Grid Database

- Controls when modified data can be seen by other transactions of the database for reading or updating [11]

3.4.4 Locking Mechanisms

Oracle also uses locks to control concurrent access to data. When updating information, the data server holds that information with a lock until the update is submitted or committed. Until that happens, no one else can make changes to the locked information. This ensures the data integrity of the system.

Oracle provides unique non-escalating row-level locking. Unlike other data servers that "escalate" locks to cover entire groups of rows or even the entire table, Oracle always locks only the row of information being updated. Because Oracle includes the locking information with the actual rows themselves, Oracle can lock an unlimited number of rows so users can work concurrently without unnecessary delays. [11]

3.4.5 Automatic Locking

Oracle locking is performed automatically and requires no user action. Implicit locking occurs for SQL statements as necessary, depending on the action requested. Oracle’s lock manager automatically locks table data at the row level. By locking table data at the row level, contention for the same data is minimized.

Oracle’s lock manager maintains several different types of row locks, depending on what type of operation established the lock. The two general types of locks are exclusive locks and share locks. Only one exclusive lock can be placed on a resource (such as a row or a table); however, many share locks can be placed on a single resource. Both exclusive and share locks always allow queries on the locked resource but prohibit other activity on the resource (such as updates and deletes). [11]
CHAPTER 4:
DISTRIBUTED TRANSACTIONS
A fundamental importance to concurrency control is the notion of a transaction. A transaction is defined as a series of actions, carried out by a single user/application program, which must be treated as an indivisible unit. [2]

A primary concern of a multiuser database management system is how to control concurrency, which is the simultaneous access of the same data by many users. Without adequate concurrency controls, data could be updated or changed improperly, compromising data integrity.[11]

Concurrency control is a method used to ensure that database transactions are executed in a safe manner (i.e., without the problem of lost update, without Violation of integrity constraints and without the problem of inconsistent retrieval). Concurrency control is especially applicable to relational databases and database management systems, which must ensure that transactions are executed safely and that they follow the ACID rules. The DBMS must be able to ensure that only serializable, recoverable schedules are allowed, and that no actions of committed transactions are lost while undoing aborted transactions.

In this chapter we will study the ACID properties and the techniques used to maintain the features of Atomicity, Consistency, Isolation and Durability. We have studied how to maintain serializability in distributed databases in order to use similar techniques to maintain serializability in a grid environment. A discussion of the Locking-based, Timestamp and Optimistic concurrency control algorithms will be presented below to show the points we have selected from the existing algorithms used in a distributed database environment and enhanced them to fit the grid database environment.

4.1 ACID:

In databases, ACID stands for Atomicity, Consistency, Isolation, and Durability. They are considered to be the key transaction processing features of a database management system, or DBMS. Without them, the integrity of the database cannot be guaranteed.

For example, a transfer of funds from one account to another is considered a transaction. A transaction might consist of multiple tasks (debiting one account and crediting another). The ACID properties guarantee that such transactions are processed reliably.

- **Atomicity** refers to the ability of the DBMS to guarantee that either all of the tasks of a transaction are performed or none of them are. The transfer of funds can be completed or it can fail for a multitude of reasons, but atomicity guarantees that one account won't be debited if the other is not credited as well.
- **Consistency** refers to the database being in a legal state when the transaction begins and when it ends. This means that a transaction can't break the rules, or what we call the integrity constraints, of the database. If an integrity constraint states that all accounts must have a positive balance, then any transaction violating this rule will be aborted.
- **Isolation** refers to the ability of the application to make operations in a transaction appear isolated from all other operations. This means that no
operation outside the transaction can ever see the data in an intermediate state; a bank manager can see the transferred funds on one account or the other, but never on both -- even if he ran his query while the transfer was still being processed. More formally, isolation means the transaction history is serializable.

- **Durability** refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone. This means it will survive system failure, and that the database system has checked the integrity constraints and won't need to abort the transaction. Typically, all transactions are written into a log that can be played back to recreate the system to the its state right before the failure. A transaction can only be deemed committed after it is safely in the log. [17]

### 4.2 Distributed Transaction:

In a distributed database environment, a transaction may access data stored at more than one site. Each transaction is divided into a number of sub-transactions, one for each site at which data accessed by the transaction is stored. These sub-transactions are represented by agents at the various sites. The agent of transaction \( T_1 \) at site \( A \) would be referred to as \( T_1^A \) and the agent of \( T_1 \) at site \( B \) would be referred to as \( T_1^B \). The distributed version of the funds transfer example is given below: [17]

![Figure 1 Distributed Transaction](image)

```
time begin transaction \( T_1 \)
    begin transaction \( T_1^A \)
        read balance_x
        balance_x = balance_x - 100
        if balance_x < 0 then
            begin
                print 'insufficient funds'
                abort \( T_1^A \)
            end
        end-if
        write balance_x
        commit \( T_1^A \)
    begin transaction \( T_1^B \)
        read balance_y
        balance_y = balance_y + 100
        write balance_y
        commit \( T_1^B \)
    commit \( T_1 \)
```

One account, say \( x \), is stored at site \( A \), while the other account \( y \), is stored at site \( B \). The indivisibility of the entire global transaction is still fundamental but in addition each sub-transaction or agent of the global transaction must be treated as an indivisible transaction by the local site at which it is executing. The sub-transactions or agents of a global transaction must not only be synchronized with other purely local concurrently executing transaction, but also with other global transactions
Concurrent Transaction Execution in Grid Database

active in the system at the same or different sites. Hence distribution adds a new dimension to the complexity of the problem of concurrency control.

Transactions can be characterized according to their duration; structure and frequency of read and write operations:
Short-duration transactions: Typical when few data items with a simple structure are involved, e.g., point queries.
Long-duration transactions: A spatial object, e.g., a polygon can consist of thousands of edges; retrieval operations (I/O) and processing of spatial joins is expensive [3]; together with human interaction and intellectual decision making on intermediate results—this implicates the presence of long duration transactions.

The problem with long transactions is that they stay for a long time in the system and occupy more resources. This results in frequent conflicts with other transactions, and often results in aborts of the long transactions. For heavy loaded systems, the long transactions very often get aborted several times before they manage to finish. The length of a transaction influences very much on the abort probabilities. For short transactions, the abort probability is rather small. For long transaction, this is not the case. Especially if we have a high number of parallel transactions, we get a lot of conflicting operations, and many aborts. [16]

Our proposed algorithm that will be detailed in chapter 5 will be very efficient to be used in the case of long transaction because a major point that was stressed in the proposed algorithm was that we should have no deadlocks or aborts due to conflicts. The major problem of aborts of long transactions will be solved with the new proposed algorithm; therefore, even if we have lots of conflicting operations, we will have no aborts.

4.3 Role of Distributed Execution Monitor

The distributed execution monitor consists of two modules: a transaction manager (TM) and a scheduler (SC). The transaction manager is responsible for coordinating the execution of the database operations on behalf of an application. The scheduler, on the other hand, is responsible for the implementation of a specific concurrency control algorithm for synchronizing access to the database.
A third component that participates in the management of distributed transactions is the local recovery managers that exist at each site. Their function is to implement the local procedures by which the local database can be recovered to a consistent state following a failure.
Each transaction originates at one site, which we will call its originating site. The execution of the database operations of a transaction is coordinated by the TM at that transaction's originating site.
The transaction managers implement an interface for the application programs which consists of five commands: begin-transaction, read, write, commit, and abort. [17]

4.4 Serializability Theory:
There are many ways in which concurrently executing transactions can interfere with one another and so compromise the integrity and consistency of the database. Examples of such interference are:

1- Lost update problem
2- Violation of integrity constraints
3- Inconsistent retrieval problem.

If the concurrent execution of transactions leaves the database in a state that can be achieved by their serial execution in some order, problems such as lost updates will be resolved. This is exactly the point of the serializability argument.

The primary function of a concurrency controller is to generate a serializable schedule for the execution of pending transactions. The issue, then, is to devise or develop algorithms that are guaranteed to generate only serializable schedules.

In a distributed databases or database grid where data is distributed on several nodes (machines) that belongs to the grid and which could be replicated, serializability theory requires more care. The schedule of transaction execution at each site is called a local schedule. If the database is not replicated, and each local schedule is serializable, their union (called the global schedule) is also serializable as long as local serialization orders are identical. In a replicated distributed database, however, the extension of the serializability theory requires more care. It is possible that the local schedules are serializable, but the mutual consistency of the database is still compromised. [17]

The corresponding breakdown of the concurrency control algorithms results in two classes:
- Those algorithms that are based on mutually exclusive access to shared data (locking), and
- Those that attempt to order the execution of the transactions according to a set of rules (protocols)

### 4.4.1 Distributed Serializability

The basic concept of serializability in the same for distributed DBMSs, but with added complexity imposed by the distribution. Consider the very simple example where two global transactions T1 and T2 each have two sub-transactions (agents) at sites A and B. Let T1A denote the agent of T1 at A, T1B the agent at B, T2A the agent of T2 at A and T2B its agent at B. Suppose that both transactions execute serially, but with T1A preceding T2A at site A, and T2B preceding T1B at site B. The schedules at site A and B, SA and SB are therefore:

SA = \[R_1(x), W_1(x), R_2(x), W_2(x)\] \(\Rightarrow\) \(T_1^A < T_2^A\)

SB = \[R_2(y), W_2(y), R_1(y), W_1(y)\] \(\Rightarrow\) \(T_2^B < T_1^B\)

Thus globally, the two transactions are not serializable even though their agents execute serially at each site.

If the global transactions T1 and T2 were launched ‘simultaneously’ by different users at sites A and B, then the schedulers operating independently at each site could schedule them in this way. Hence, for distributed transactions, we require serializability of all local schedules (both purely local and local agents of global
transactions) and global serializability for all global transactions. Effectively, this means that all sub-transactions of global transactions appear in the same order in the equivalent serial schedule at all sites, that is

if \( T_i^A < T_j^A \)

Then \( T_i^K < T_j^K \) for all sites \( K \) at which \( T_i \) and \( T_j \) have agents.

\( T_1^K < T_2^K < T_3^K < T_4^K < \ldots < T_n^K \) is known as the local ordering for site \( K \)

While

\( T_1 < T_2 < T_3 < T_4 < \ldots < T_n \) is known as the global ordering for all sites.

With distributed transaction control, the following sequence of steps is required to process a global transaction:

**Begin:**

Step 1: A global transaction is initiated at site \( A \) via the global transaction manager (GTMA)

Step 2: Using information about the location of data (from the catalogue or data dictionary, the GTMA divides the global transaction into a series of agents at each relevant site

Step 3: The global communications manager (GCM\( ^A \)) at \( A \) sends these agents to the appropriate sites via the communications network

Step 4: Once all agents have completed, the results are communicated back to site \( A \) via the GCMs.

**End** [17]

### 4.5 CONCURRENCY CONTROL TECHNIQUES

There are three basic concurrency control techniques which allow transactions to execute safely in parallel subject to certain constraints:

(1) Locking methods

(2) Timestamp methods

(3) Optimistic methods

Both locking and timestamping are essentially conservative approaches in that they cause transactions to be delayed in case they may conflict with other transactions at some time in the future. Optimistic methods are based on the premise that conflict is rare and allow transactions to proceed unsynchronized, and only check for conflicts at the need when a transaction commits.
4.5.1 LOCKING-BASED CONCURRENCY CONTROL ALGORITHMS

The main idea of locking-based concurrency control is to ensure that the data that is shared by conflicting operations is accessed by one operation at a time. This is accomplished by associating a "lock" with each lock unit (some portion or granule of the database). This lock is set by a transaction before it is accessed and is reset at the end of its use. Obviously a lock unit cannot be accessed by an operation if it is already locked by another. Thus a lock request by a transaction is granted only if the associated lock is not being held by any other transaction.

There are two types of locks (commonly called lock modes) associated with each lock unit: read lock (rl) and write lock (wl). A transaction Ti that wants to read a data item contained in lock unit x obtains a read lock on x [denoted rli(x)]. The same happens for write operations. It is common to talk about the compatibility of lock modes. Two lock modes are compatible if two transactions which access the same data item can obtain these locks on that data item at the same time. Therefore it is possible for two transactions to read the same data item concurrently.

In locking-based systems, the scheduler is a lock manager (LM). The transaction manager passes to the lock manager the database operation (read or write) and associated information (such as the item that is accessed and the identifier of the transaction that issues the database operation). The lock manager then checks if the lock unit that contains the data item is already locked. If so, and if the existing lock mode is incompatible with that of the current transaction, the current operation is delayed. Otherwise, the lock is set in the desired mode and the database operation is passed on to the data processor for actual database access. The transaction manager is then informed of the results of the operation. The termination of a transaction results in the release of its locks and the initiation of another transaction that might be waiting for access to the same data item.

The locking algorithm will not unfortunately properly synchronize transaction executions. This is because to generate serializable schedules, the locking and releasing operations of transactions also need to be coordinated.

The locking algorithm releases the locks that are held by a transaction (say, Ti) as soon as the associated database command (read or write) is executed, and that lock unit (say x) no longer needs to be accessed. However, the transactions itself is locking other items (say, y), after it releases its lock on x. Even though this may seem to be advantageous from the viewpoint of increased concurrency, it permits transactions to interfere with one another, resulting in the loss of total isolation and atomicity. Hence the argument for two-phase locking (2PL).

The two-phase locking rule simply states that no transaction should request a lock after it releases one of its locks. Alternatively, a transaction should not release a lock until it is certain that it will not request another lock. 2PL algorithms execute transactions in two phases. Each transaction has a growing phase, where it obtains locks and accesses data items, and a shrinking phase, during which it releases locks. The lock point is the moment when the transaction has achieved all its locks but has not yet started to release any of them. Thus the lock point determines the end of the growing phase.
4.5.1.1 Centralized 2 PL:

The 2PL algorithm can be extended to the (replicated or partitioned) distributed DBMS environment. One way of doing this is to delegate lock management responsibility to a single site only. This means that only one of the sites has a lock manager; the transaction managers at the other sites communicate with it rather than with their own lock managers. This approach is also known as the primary site 2PL algorithm.

The communication between the cooperating sites in executing a transaction according to a centralized 2PL (C2PL) algorithm is depicted in the figure below:

![Figure 2 Communication Structure of Centralized 2PL](image)

This communication is between the transaction manager at the site where the transaction is initiated (called the coordinating TM), the lock manager at the central site, and the data processors (DP) at the other participating sites. The participating sites are those at which the operation is to be carried out. The order of messages is denote in the figure.

One common criticism of C2PL algorithms is that a bottleneck may quickly form around the central site. Furthermore, the system may be less reliable since the failure or inaccessibility of the central site would cause major system failures. There are studies that indicate that the bottleneck will indeed form as the transaction rate increases, but is insignificant at low transaction rates. Furthermore, sharp performance degradation at high loads is observed in other locking-based algorithm as well.[17]
4.5.1.2 Primary Copy 2PL:

Primary copy 2PL (PC2PL) is a straightforward extension of centralized 2PL in an attempt to counter the latter's potential performance problems, discussed above. Basically, it implements lock managers at a number of sites and makes each lock manager responsible for managing the locks for a given set of lock units. The transaction managers then send their lock and unlock requests to the lock managers that are responsible for that specific lock unit. Thus the algorithm treats one copy of each data item as its primary copy.

The changes of the primary copy 2PL algorithm are minimal according to the centralized 2PL. Basically, the only change is that the primary copy locations have to be determined for each data item prior to sending a lock or unlock request to the lock manager at that site.

Primary copy 2PL was proposed for the prototype distributed version of INGRES. Even though it demands a more sophisticated directory at each site, it also reduces the load of the central site without causing a large amount of communication among the transaction managers and lock managers. In one sense it is an intermediate step between the centralized 2PL that we discussed in the preceding section and the distributed 2PL in the next section.

4.5.1.3 Distributed 2PL:

Distributed 2PL (D2PL) expects the availability of lock managers at each site. If the database is not replicated, distributed 2PL degenerates into the primary copy 2PL algorithm. If the data is replicated, the transaction implements the ROWA replica control protocol.

The communication between cooperating sites that execute a transaction according to the distributed 2PL protocol is depicted in the figure below. Notice that Figure does not show application of the ROWA rule.

![Figure 3 Communication Structure of Distributed 2PL](image)

The distributed 2PL transaction management algorithm is similar to the C2PL-TM, with two major modifications. The messages that are sent to the central site lock manager in C2PL-TM are sent to the lock managers at all participating sites in the
D2PL-TM. The second difference is that the operations are not passed to the data processors by the coordinating transaction manager, but by the participating lock managers. This means that the coordinating transaction manager does not wait for a "lock request granted" message. Another point about the above figure is the following. The participating data processors send the "end of operation" message to the coordinating TM. The alternative is for each DP to send it to its own lock manager who can then release the locks and inform the coordinating TM. [17]

4.5.2 TIMESTAMP-BASED CONCURRENCY CONTROL ALGORITHMS

Timestamp methods of concurrency control are quite different from locking methods. Locking generally involve transactions which make conflicting requests wait. With timestamp methods, on the other hand, there is no waiting; transactions involved in conflict are simply rolled back and restarted.

Unlike the locking-based algorithms, timestamp-based concurrency control algorithms do not attempt to maintain serializability by mutual exclusion. Instead, they select, a priori, a serialization order and execute transactions accordingly. To establish this ordering, the transaction manager assigns each transaction T, a unique timestamp, ts(T), at its initiation.

A timestamp is a simple identifier that serves to identify each transaction uniquely and to permit ordering. Uniqueness is only one of the properties of timestamp generation. The second property is monotonicity. Two timestamps generated by the same transaction manager should be monotonically increasing. Thus timestamps are values derived from a totally ordered domain. It is this second property that differentiates a timestamp from a transaction identifier. [17]

There are a number of ways that timestamps can be assigned. One method is to use a global (system wide) monotonically increasing counter. However, the maintenance of global counters is a problem in distributed systems. Therefore, it is preferable that each site autonomously assign timestamps based on its local counter. To maintain uniqueness, each site appends its own identifier to the counter value. Thus the timestamp is a two-tuple of the form <local counter value, site identifier>. Note that the site identifier is appended in the least significant position. Hence it serves only to order the timestamps of two transactions that might have been assigned the same local counter value. If each system can access its own system clock, it is possible to use system clock values instead of counter values.

With this information, it is simple to order the execution of the transactions’ operations according to their timestamps. Formally, the timestamp ordering (TO) rule can be specified as follows:

TO Rule: Given two conflicting operations O, and O, belonging, respectively, to transactions T, and T, O, is executed before O, if and only if ts(T,) < ts(T,). In this case T, is said to be the older transaction and T, is said to be the younger one.

A scheduler that enforces the TO rule checks each new operation against conflicting operations that have already been scheduled. If the new operation belongs to a transaction that is older than all the conflicting ones that have already been scheduled, the operation is accepted; otherwise, it is rejected, causing the entire transaction to restart with a new timestamp.
A timestamp ordering scheduler that operates in this fashion is guaranteed to generate serializable scheduler. However, this comparison between the transaction timestamps can be performed only if the scheduler has received all the operations to be scheduled. If operations come to the scheduler one at a time (which is a realistic case), it is necessary to be able to detect if an operation has arrived out of sequence. To facilitate this check, each data item x is assigned two timestamps: a read timestamp \([\text{rts}(x)]\), which is the largest of the timestamps of the transactions that have read x, and a write timestamp \([\text{wts}(x)]\), which is the largest of the timestamps of the transactions that have written (updated) x. It is now sufficient to compare the timestamp of an operation with the read and write timestamps of the data item that it wants to access to determine if any transaction with a larger timestamp has already accessed the same data item.

The transaction manager is responsible for assigning a timestamp to each new transaction and attaching this timestamp to each database operation that it passes on to the scheduler. The latter component is responsible for keeping track of read and write timestamps as well as performing the serializability check. [17]

### 4.5.2.1 Basic TO Algorithm:

The basic TO algorithm is a straightforward implementation of the TO rule. The coordinating transaction manager assigns the timestamp to each transaction, determines the sites where each data item is stored, and sends the relevant operations to these sites. The schedulers at each site simply enforce the TO rule.

A transaction which contains an operation that is rejected by a scheduler is restarted by the transaction manager with a new timestamp. This ensures that the transaction has a chance to execute in its next try. Since the transactions never wait while they hold access rights to data items, the basic TO algorithm never causes deadlocks. However, the penalty of deadlock freedom is potential restart of a transaction numerous times.

Another detail that needs to be considered relates to the communication between the scheduler and the data processor. When an accepted operation is passed on to the data processor, the scheduler needs to refrain from sending another incompatible, but acceptable operation to the data processor until the first is processed and acknowledged. This is a requirement to ensure that the data processor executes the operations in the order in which the scheduler passes them on. Otherwise, the read and write timestamps values for the accessed data item would not be accurate. [17]

**Example:**
Assume that the TO scheduler first receives \(W(x)\) and then receives \(W_j(x)\), where \(ts(T_1) < ts(T_j)\). The scheduler would accept both operations and pass them on to the data processor. The result of these two operations is that \(wts(x) = ts(T_j)\), and we then expect the effect of \(W_j(x)\) to be represented in the database. However, if the data processor does not execute them in that order, the effects on the database will be wrong.

The scheduler can enforce the ordering by maintaining a queue for each data item that is used to delay the transfer of the accepted operation until an acknowledgment is received from the data processor regarding the previous operation on the same data item.

Such a complication does not arise in 2PL-based algorithms because the lock manager effectively orders the operations by releasing the locks only after the
operation is executed. In one sense the queue that the TO scheduler maintains may be thought of as a lock. However, this does not imply that the schedule generated by a TO scheduler and a 2PL scheduler would always be equivalent. There are some schedules that a TO scheduler would generate that would not be admissible by a 2PL schedule.

Remember that in the case of strict 2PL algorithms, the releasing of locks is delayed further, until the commit or abort of a transaction. It is possible to develop a strict TO algorithm by using a similar scheme. For example, if $W_i(x)$ is accepted and released to the data processor, the scheduler delays all $R_j(x)$ and $W_j(x)$ operations (for all $T_j$) until $T_i$ terminates (commits or aborts).

### 4.5.2.2 Conservative TO Algorithm:

The conservative TO algorithms attempt to lower this system overhead by reducing the number of transaction restarts.

The basic TO algorithm tries to execute an operation as soon as it is accepted. It is therefore "aggressive" or "progressive". Conservative algorithms, on the other hand, delay each operation until there is an assurance that no operations with a smaller timestamp can arrive at the scheduler. If this condition can be guaranteed, the scheduler will never reject an operation. However this delay introduces the possibility of deadlocks.

The basic technique that is used in conservative TO algorithms is based on the following idea: the operations of each transaction are buffered until an ordering can be established so that rejections are not possible, and they are executed in that order.

Assume that each scheduler maintains one queue for each transaction manager in the system. The scheduler at site $i$ stores all the operations that it receives from the transaction manager at site $j$ in queue $Q_{ij}$. Scheduler $i$ has one such queue for each $j$. When an operation is received from a transaction manager, it is placed in its appropriate queue in increasing timestamp order. The schedulers at each site execute the operations from these queues in increasing timestamp order.

This scheme will reduce the number of restarts, but it will not guarantee that they will be eliminated completely. Consider the case where at site $i$ the queue for site $j$ ($Q_{ij}$) is empty. The scheduler at site $i$ will choose an operation (say, $R(x)$) with the smallest timestamp and pass it on the data processor. However, site $j$ may have sent to $i$ an operation (say, $W(x)$) with a smaller timestamp which may still be in transit in the network. When this operation reaches site $i$, it will be rejected since it violates the TO rule: it wants to access a data item that is currently being accessed (in an incompatible mode) by another operation with a higher timestamp.

It is possible to design an extremely conservative TO algorithm by insisting that the scheduler choose an operation to be sent to the data processor only if there is at least one operation in each queue. This guarantees that every operation that the scheduler receives in the future will have timestamps greater than or equal to those currently in the queues. Of course, if a transaction manager does not have a transaction to process; it needs to send dummy messages periodically to every scheduler in the system, informing them that the operations that it will send in the future will have timestamps greater than that of the dummy message.
The extremely conservative timestamp ordering scheduler actually executes transactions serially at each site. This is very restrictive. One method that has been employed to overcome this restriction is to group transactions into classes. Transaction classes are defined with respect to their read sets and write sets. It is therefore sufficient to determine the class that a transaction belongs to by comparing to transaction’s read set and write set, respectively, with the read set and write set of each class. Thus the conservative TO algorithm can be modified so that instead of requiring the existence, at each site, of one queue for each transaction manager, it is only necessary to have one queue for each transaction class. Alternatively, one might mark each queue with the class to which it belongs. With either of these modifications, the conditions for sending an operation to the data processor are changed. It is no longer necessary to wait until. There is at least one operation in each class to which the transaction belongs. This and other weaker conditions that reduce the waiting delay be defined and are sufficient. [17]

4.5.2.3 Multiversion TO Algorithm

Multiversion TO is another attempt at eliminating the restart overhead cost of transactions, most of the work on multiversion TO has concentrated on centralized databases, so we present only a brief overview. However we should indicate that the multiversion TO algorithm would be a suitable concurrency control mechanism for DBMSs that are designed to support applications which inherently have a notion of versions of database objects (e.g. engineering databases and document databases).

In multiversion TO, the updates do not modify the database; each write operation creates a new version of the data item. Each version is marked by the timestamp of the transaction that creates it. Thus the multiversion TO algorithm trades storage space for time. In doing so, it processes each transaction on a state of the database that is would have seen if the transactions were executed serially in timestamp order.

The existence of versions is transparent to users who issue transactions simply by referring to data items, not to any specific version. The transaction manager assigns a timestamp to each transaction which is also used to keep track of the timestamps of each version. The operations are processed by the schedulers as follows:

- A R₁(x) is translated into a read on one version of x. This is done by finding a version of x (say, xₙ) such that ts(xₙ) is the largest timestamp less than ts(T₁). R₁(xₙ) is then sent to the data processor.
- A W₁(x) is translated into W₁(xₙ) so that ts(xₙ) = ts(T₁) are sent to the data processor if and only if no other transaction with a timestamp greater than ts(T₁) has read the value of a version of x (say, xₙ) such that ts(xₙ) > ts(xₙ). In other words, if the scheduler has already processed a R₁(xₙ) such that ts(T₁) < ts(xₙ) < ts(T₂)

Then W₁(x) is rejected.

A scheduler that processes the read and the write requests of transactions according to the rules noted above is guaranteed to generate serializable schedules. To save space, the versions of the database may be purged from time to time. This should be done when the distributed DBMS is certain that it will no longer receive a transaction that needs to access the purged version.
4.5.3 OPTIMISTIC CONCURRENCY CONTROL ALGORITHMS

The concurrency control algorithms discussed earlier are pessimistic in nature. In other words, they assume that the conflicts between transactions are quite frequent and do not permit a transaction to access a data item if there is a conflicting transaction that accesses that data item. Thus the execution of any operation of a transaction follows the sequence of phases: validation (V), read (R), computation (C), and write (W). Generally, this sequence is valid for an update transaction as well as for each of its operations. [17]

| Validate | Read | Compute | Write |

Figure 4 Phases of Pessimistic Transaction Execution

Optimistic algorithms, on the other hand, delay the validation phase until just before the write phase. Thus an operation submitted to an optimistic scheduler is never delayed. The read, compute, and write operations of each transaction are processed freely without updating the actual database. Each transaction initially makes its updates on local copies of data items. The validation phase consists of checking if these updates would maintain the consistency of the database. If the answer is affirmative, the changes are made global (i.e., written into the actual database). Otherwise the transaction is aborted and has to restart.

| Read | Compute | Validate | Write |

Figure 5 Phase of Optimistic Transaction Execution

Each transaction $T_i$ is subdivided (by the transaction manager at the originating site) into a number of subtransactions, each of which can execute at many sites. Notationally, let us denote by $T_{ij}$ a subtransaction of $T_i$ that executes at site $j$. Until the validation phase, each local execution follows the sequence depicted in the figure above. At that point a timestamp is assigned to the transaction which is copied to all its subtransactions. The local validation of $T_{ij}$ is performed according to the following rules, which are mutually exclusive.

Rule 1: If all transactions $T_k$ where $ts(T_k) < ts(T_{ij})$ have completed their write phase before $T_{ij}$ has started its read phase, validation succeeds, because transaction executions are in serial order.

Rule 2: If there is any transaction $T_k$ such that $ts(T_k) < ts(T_{ij})$ which completes its write phase while $T_{ij}$ is in its read phase, the validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$

Rule 3: If there is any transaction $T_k$ such that $ts(T_k) < ts(T_{ij})$ which completes its write phase before $T_{ij}$ completes its read phase, the validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$, and $WS(T_k) \cup WS(T_{ij}) = \emptyset$
Concurrent Transaction Execution in Grid Database

Rule 1 is obvious; it indicates that the transactions are actually executed serially in their timestamp order. Rule 2 ensures that none of the data items updates by T_k are read by T_ij and that T_k finishes writing its updates into the database before T_ij starts writing. Thus the updates of T_k will not be overwritten by the updates of T_k. Rule 3 is similar to Rule 2, but does not require that T_k finish writing before T_ij starts writing. It simply requires that the updates of T_k not affect the read phase of the write phase of T_ij.

Once a transaction is locally validated to ensure that the local database consistency is maintained, it also needs to be globally validated to ensure that the mutual consistency rule is obeyed. Unfortunately, there is no known optimistic method of doing this. A transaction is globally validated if all the transactions that precede it in the serialization order (at that site) terminate (either by committing or aborting). This is a pessimistic method since it performs global validation early and delays a transaction. However, it guarantees that transactions execute in the same order at each site.

An advantage of the optimistic concurrency control algorithms is their potential to allow a higher level of concurrency. It has been shown that when transaction conflicts are very rare, the optimistic mechanism performs better than locking. A major problem with optimistic algorithms is the higher storage cost. To validate a transaction, the optimistic mechanism has to store the read and the write sets of several other terminated transactions. Specifically, the read and write sets of terminated transactions that were in progress when transaction T_ij arrived at site j need to be stored in order to validate T_ij. Obviously, this increases the storage cost.
CHAPTER 5:
PROPOSED CONCURRENT TRANSACTION ALGORITHM
We tried to identify the characteristics and requirements of data grids and how they can be met in most efficient way. Special attention is given to transaction concurrency in a database grid as also as it is stated in the following references [8, 19].

We have noticed that due to the large amount of data we should consider in grid architecture environment, the algorithm of transaction concurrency control should satisfy the following:

1- The size of the data should be taken into account;
2- We should have no deadlocks or aborts due to conflicts.
3- Since the host nodes are by large stateless and require very little persistency. If a process fails its active transactions (which are un-committed by definition) are cancelled and eventually restarted. So we have to take into account the transient hosts in a grid computing environment.
4- We should have minimal reads and writes to disk storage. Calculations are likely to be satisfied by cached records.
5- The algorithm must be suitable for distributed execution.

Therefore, we need to incorporate, in this algorithm, techniques which are suitable for execution in a grid of transient hosts.

5.1 Proposed Algorithm:

As concurrency control algorithms in centralized databases were used and enhanced towards distributed databases, we will be following the same trajectory in order to develop concurrency control algorithms for Grid Database.

In our Opinion, existing algorithms that could be extended to control concurrent transactions into a database grid could be based on Timestamp algorithms and Optimistic algorithms due to the following reasons:

Timestamp methods of concurrency control are quite different from locking methods. No locks are involved and there can therefore be no deadlock. Locking methods generally involve transactions which make conflicting requests wait. With the timestamp methods, on the other hand, there is no waiting; transactions involved in conflict are simply rolled back and restarted.

On the other hand, the Optimistic methods of concurrency control which are based on the premise that conflict is rare and that the best approach is to allow transactions to proceed unimpeded by complex synchronization methods and without any waiting. When a transaction wishes to commit, the system then checks for conflicts and in the event of a conflict being detected restarts the transaction.

The algorithm should synchronize multiple concurrent transactions executing on distributed hosts or nodes of the grid. It is suitable for execution in an environment of numerous transient hosts connected over a (relatively) high latency network.
Scheme of the proposed algorithm: (see Figure 6)

- A Time stamp is assigned to all transactions. A strong serialization order is maintained by all transactions.

- Transactions begin to execute immediately (optimistically) on any host but do not commit.

- As a transaction is executed it distributes the update messages to all subsequent transactions with a younger time stamp order. These messages contain iterative updates made so far by the transaction and their time cost is negligible.

- When a transaction commits it sends a final message. This enables the next transaction (by time stamping order) to commit.

- A later transaction which begins to execute concurrently accumulates the messages from earlier transactions.

- This transaction checks for any conflict with earlier transactions and if it can determine that it is not in conflict with any earlier transaction, it may safely commit.

- It too sends a final message.

- A later transaction that has detected a conflict with earlier transactions continues to execute optimistically but does not commit. In order to resolve the conflicts detected it may use incoming updates from the messages already accumulated to backtrack (undo and redo) and recalculate when necessary or possible. So it is necessary to undo all the write operations made so far until the conflicting operation(s) then continue all the way down with re-execution of the operations of the transaction.

- When a final message is finally received the transaction should be able to commit faster because at that time no other messages will be received and no other conflicts will be caused.
Concurrent Transaction Execution in Grid Database

Transaction 1:
READ X
WRITE X

Transaction 2:
READ X, Y
WRITE Y

Transaction 3:
READ Y, Z
WRITE Z

Executing

1st message By 1 using record X

1st message By 2 using record Y

Begin Optimistic Execution

Begin Optimistic Execution

2nd message By 1 -Provisional- Updated X values

2nd message By 2 -Provisional- Updated Y values

Committing

Final message By 1 No further changes

Concurrent commit by both transactions

Figure 6 Cascading Commit of Concurrent transactions
A transaction that executes optimistically continues to send out provisional-update messages. These messages contain changes which may be committed, but are in fact provisional on the outcome of previous transactions. Other transactions can therefore also determine provisional conflicts. Consequently, a single final message may resolve multiple dependencies in multiple transactions and start a cascading tree of commits. This is particularly likely with a complex update transaction which delays many other transactions.

With the design of the Grid outlined in the previous section, several properties are apparent:

1- There are no deadlocks or aborts due to conflicts
2- The host nodes are by large stateless and requires very little persistency. If a process fails its active transactions (which are un-committed by definition) are cancelled and eventually restarted.
3- Records are read and written to disk store precisely one. Any recalculations are likely to be satisfied by the cached records or data from the messages.
4- It is very suitable to distributed execution.

5.2 Scenario of Execution

Consider we have three concurrent transactions which are: Transaction 1, Transaction 2 and Transaction 3.

The list of data items that each transaction will be accessing are: Transaction 1 will be accessing the data item X, Transaction 2 will be accessing the data items X and Y and Transaction 3 will be accessing the data items X, Y and Z

The details of each transaction are as follows (operations):

Transaction 1 has the older timestamp order and its operations are:
READ X
WRITE X = X+1

Transaction 2 has the second older timestamp order and its operations are:
READ X
READ Y
WRITE Y = X+Y

Transaction 3 is the younger Transactions and has the following operations:
READ Y
READ Z
WRITE Z = Y+Z

The initial values of the data items are:
X=4
Y=20
Z=45
In the first scenario described below we have presented the three transactions executing serially, then we have presented the scenario concerning the proposed algorithm where the three transactions are executed concurrently where the messages are sent to all younger concurrent transactions only when making updates.

### 5.2.1 Serial Execution Scenario

Serial Execution of transactions with no concurrent access to the same data item:

<table>
<thead>
<tr>
<th>Transaction 1 (1\textsuperscript{st} TO)</th>
<th>Transaction 2 (2\textsuperscript{nd} TO)</th>
<th>Transaction 3 (3\textsuperscript{rd} TO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>READ (X=4)</td>
<td>READ (X=5)</td>
<td>READ (Y=25)</td>
</tr>
<tr>
<td>WRITE (X=X+1=5)</td>
<td>READ (Y=20)</td>
<td>READ (Z=45)</td>
</tr>
<tr>
<td>COMMIT (T1)</td>
<td>WRITE (Y=X+Y=25)</td>
<td>WRITE (Z=Y+Z=70)</td>
</tr>
<tr>
<td></td>
<td>COMMIT (T2)</td>
<td>COMMIT (T3)</td>
</tr>
</tbody>
</table>

Figure 7 Serial Execution
5.2.2 Concurrent Execution Scenario

This is a scenario of Concurrent Execution of transactions where update messages are sent to concurrent transactions only after a write operation made to the database in order to minimize the flow of update message to concurrent transactions.

**Transaction 1 (1\textsuperscript{st} TO)**

- READ $X=4$
- WRITE $X=X+1=5$

**Transaction 2 (2\textsuperscript{nd} TO)**

- READ $X=4$
- READ $Y=20$
- Message received from T1, Provisional Update, $X=5$
- Backtrack and re-read $X$
- READ $X=5$
- READ $Y=20$
- WRITE $Y=X+Y=25$

**Transaction 3 (3\textsuperscript{rd} TO)**

- READ $Y=20$
- READ $Z=45$
- Message received from T1, Provisional Update, $X=5$
- Message received from T2, Provisional Update, $Y=25$
- Backtrack and re-read $Y$
- READ $Y=25$
- READ $Z=45$
- WRITE $Z=Y+Z=70$

**Figure 8 Concurrent Execution Scenario**

- COMMIT (T1)
- Final message received from T1
- COMMIT (T2)
- Final message received from T1
- Final message received from T2
- COMMIT (T3)
5.3 Comparison between the timestamp based algorithms and the proposed algorithm

The timestamp based algorithms select, a priori, a serialization order and execute transactions accordingly. To establish this ordering, the transaction manager assigns each transaction \( T_i \) a unique timestamp, \( ts(T_i) \), at its initiation.

We have the scheduler that enforces the TO rule and checks each new operation against conflicting operations that have already been scheduled and which belong to older transactions. If the new operation belongs to a transaction that is older than all the conflicting ones that have already been scheduled, the operation is accepted; otherwise, it is rejected, causing the entire transaction to restart with a new timestamp.

In the proposed new concurrency control algorithm, also each transaction \( T_i \) has a unique timestamp, \( ts(T_i) \), at its initiation. The transaction continues to execute and just before the commit it checks if there are conflicts with earlier operations therefore, the transaction backtracks using the accumulated messages until no more conflicts are detected and it is never restarted with a new timestamp.

Since the transactions never wait while they hold access rights to data items, the basic TO algorithm never causes deadlocks.
However, the penalty of deadlock freedom is potential restart of a transaction numerous times.
But in the proposed new algorithm, no restart of a transaction will occur because we used the concept of backtracking of the operations in conflict.

We have used the Basic TO algorithm because even though the Conservative TO Algorithm and the Multiversion TO Algorithm were developed to minimize or eliminate the restart overhead cost but there were still limitations.

The Conservative TO Algorithm will reduce the number of restarts by delaying each operation until there is an assurance that no operation with a smaller timestamp can arrive at the scheduler. However this delay introduces the possibility of deadlocks. Other techniques such as queue of transactions' operations are used to reduce the waiting delay.

The multiversion TO Algorithm has mostly concentrated on centralized database. In multiversion TO, the updates do not modify the database; each write operation creates a new version of the data item. Each version is marked by the timestamp of the transaction that creates it. Thus the multiversion TO algorithm trades storage space for time. In doing so, it processes each transaction on a state of the database that is would have seen if the transactions were executed serially in timestamp order.
5.4 Comparison between the Optimistic based algorithm and the proposed algorithm:

In the new proposed algorithm transactions begin to execute immediately (optimistically) on any host but do not commit just like in Optimistic algorithms, where the delay of the validation phase until just before the write phase, in order to check for conflicts just before the commit. Thus an operation submitted to an optimistic scheduler is never delayed the concept that was inherited for the new proposed concurrency control algorithm.

In the Optimistic algorithm, the validation phase consists of checking if these updates would maintain the consistency of the database. If the answer is affirmative, the changes are made global that is written into the actual database. Otherwise the transaction is aborted and has to restart.

Whereas in the new proposed concurrency control algorithm, the transaction is not aborted but it backtracks to resolve the conflicts of the operation by using the read and write sets of several terminated transactions or more precisely of the older transactions by time stamping order that has already committed.

In the Optimistic based algorithm we used to worry about the increases of the storage cost. But in a data grid environment and with the grid design presented earlier we have the distributed caching that is a main memory of all the connected nodes of the grid which can increase the storage and the read performance with low cost.
5.5 Summary Tables

Let us consider the cost in our example for the READ operation is 1 unit of time, WRITE is 2 units, COMMIT is 2 units and UNDO is 3 units. Through measuring the time of execution of concurrent transactions between the three algorithms and as shown below the proposed algorithm will achieve a higher concurrency and it is also obvious that it is very suitable for concurrent execution of long transactions.

Notice that the operations of Transaction 1 are:
- READ U
- READ W
- WRITE W=W+U
- READ X
- READ WRITE X=X+1
- COMMIT (T1)

The operations of Transaction 2 are:
- READ U
- READ V
- WRITE V=V+U
- READ X
- WRITE Y=Y+X
- COMMIT (T2)

5.5.1 Timestamp Algorithm:

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
<th>Timing (Unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ U</td>
<td>READ U</td>
<td>1</td>
</tr>
<tr>
<td>READ W</td>
<td>READ V</td>
<td>1</td>
</tr>
<tr>
<td>WRITE W=W+U</td>
<td>WRITE V=V+U</td>
<td>2</td>
</tr>
<tr>
<td>READ X=4</td>
<td>READ X=4</td>
<td>1</td>
</tr>
<tr>
<td>WRITE X=5</td>
<td>READ Y=20</td>
<td>2</td>
</tr>
<tr>
<td>COMMIT (T1)</td>
<td>ABORT &amp; RESTART</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>UNDO WRITE V</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>READ U</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>READ V</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>WRITE V=V+U</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>READ X=5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>READ Y=20</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>WRITE Y=Y+X=25</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>COMMIT (T2)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>22</td>
</tr>
</tbody>
</table>

Figure 9 Timestamp Algorithm
### 5.5.2 Optimistic Algorithm:

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
<th>Timing (Unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ U</td>
<td>READ U</td>
<td>1</td>
</tr>
<tr>
<td>READ W</td>
<td>READ V</td>
<td>1</td>
</tr>
<tr>
<td>WRITE W=W+U</td>
<td>WRITE V=V+U</td>
<td>2</td>
</tr>
<tr>
<td>READ X=4</td>
<td>READ X=4</td>
<td>1</td>
</tr>
<tr>
<td>WRITE X=X+1=5</td>
<td>READ Y=20</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>WRITE Y=Y+X=25</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>ABORT &amp; RESTART</td>
<td></td>
</tr>
<tr>
<td>COMMIT (T1)</td>
<td>UNDO WRITE Y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>UNDO WRITE V</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>READ U</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>READ V</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>WRITE V=V+U</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>READ X=5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>READ Y=20</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>WRITE Y=Y+X=25</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>COMMIT (T2)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>27</td>
</tr>
</tbody>
</table>

Figure 10 Optimistic Algorithm

### 5.5.3 Proposed Algorithm:

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
<th>Timing (Unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ U</td>
<td>READ U</td>
<td>1</td>
</tr>
<tr>
<td>READ W</td>
<td>READ V</td>
<td>1</td>
</tr>
<tr>
<td>WRITE W=W+U</td>
<td>WRITE V=V+U</td>
<td>2</td>
</tr>
<tr>
<td>READ X=4</td>
<td>READ X=4</td>
<td>1</td>
</tr>
<tr>
<td>WRITE X=X+1=5</td>
<td>READ Y=20</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Backtrack READ X=5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>READ Y =20</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>WRITE Y+X=25</td>
<td>2</td>
</tr>
<tr>
<td>COMMIT (T1)</td>
<td>COMMIT (T2)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 11 Proposed Algorithm
5.5.4 Further Examples and Testing:

We have taken all the possibilities of execution of two transactions $T_1$ and $T_2$ accessing one same data item $a$, and we studied the time of execution following the three algorithms: Timestamp, Optimistic and Proposed. The execution of these transactions is as follows:

The Non-Conflicted executions of transactions that satisfy the ACID properties are:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
</tr>
<tr>
<td>write (a)</td>
<td>write (a)</td>
<td>read (a)</td>
<td>read (a)</td>
</tr>
</tbody>
</table>

The conflicted executions of transactions that do not satisfy the ACID properties are:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
</tr>
<tr>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
</tr>
</tbody>
</table>

The comparison of the performance for these scenarios of execution between the three algorithms is the same and as following:

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Optimistic</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>12</td>
<td>7</td>
</tr>
</tbody>
</table>

Timestamp took 12 units of time, the Optimistic took 12 units of time and the proposed algorithm took 7 units of time, which shows the efficiency of the new proposed algorithm versus the Timestamp and Optimistic algorithms.

We have also taken all the possibilities of execution of two transactions $T_1$ and $T_2$ accessing two data items $a$ and $b$, and we studied the time of execution following the three algorithms: Timestamp, Optimistic and Proposed. The execution of these transactions is as follows:

The Non-Conflicted executions of transactions that satisfy the ACID properties are:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
<td>read (a)</td>
</tr>
<tr>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
<td>write (a)</td>
</tr>
</tbody>
</table>

-48-
Since there are no conflicts in the above possible scenarios of execution, so the same time is taken by the three algorithms.

The conflicted executions of transactions that do not satisfy the ACID properties and where the conflict is towards the middle of the transactions are:
In all these scenarios of execution, the comparison of the performance between the three algorithms is the same and is as following:

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Optimistic</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>24</td>
<td>13</td>
</tr>
</tbody>
</table>

Timestamp took 18 units of time, the Optimistic took 24 units of time and the proposed algorithm took 13 units of time, which shows the efficiency of the new proposed algorithm versus the Timestamp and Optimistic algorithms.
The conflicted executions of transactions that do not satisfy the ACID properties and where the conflict is toward the end of the transaction and which are the most costly in timestamp and Optimistic algorithms:

The comparison of the performance between the three algorithms in these cases is the same and as following:

<table>
<thead>
<tr>
<th></th>
<th>Timestamp</th>
<th>Optimistic</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>24</td>
<td>24</td>
<td>13</td>
</tr>
</tbody>
</table>

Timestamp took 24 units of time, the Optimistic took 24 units of time and the proposed algorithm took 13 units of time, which again shows the efficiency of the new proposed algorithm versus the Timestamp and Optimistic algorithms.
CHAPTER 6:
CONCLUSION AND FUTURE RESEARCH
Grid computing has attracted much attention in the last few years and we expect that it will continue to gain popularity. However, while Grid computing has great potential, many issues such as reliability, efficient replica management, efficient concurrent transaction execution and security must be resolved before Grid computing can be widely used and accepted.

There appears to be no reason why Grid applications will not require at least the same functionalities, tools and properties as other types of database applications. Consequently, the range of facilities already offered by existing DBMSs will be required and enhanced in order to support the grid technology architecture. The same as distributed databases used and enhanced existing concurrency control algorithms of centralized databases.

In my thesis I have proposed a new transaction concurrency algorithm, which is based on the combination of two existing algorithms in distributed databases; Timestamp and Optimistic algorithms. This new algorithm is CPU- and network-intensive, but reduces contention for storage resources and is suitable for execution in grids where nodes may be transient. I have also shown that the proposed algorithm is good for concurrent execution of long transaction.

Some disadvantages concerning the proposed algorithm can be listed:

The flow of messages between transactions should be studied in future research and especially if we have a large number of concurrent transactions accessing a large number of data items, in order to allow younger transactions to process correctly all the incoming messages.

We have considered that the time taken to send and receive the update message by the transactions is negligible. This time taken to process the update messages should be also considered in future research to see if it impacts in any way the performance of the transaction execution.

We have considered in the proposed algorithm that transactions are time stamped by a shared/global counter and this also should be carried in the future research especially we might have a great number of nodes in the grid that could be either worker or storage nodes.

The examples that were given in the previous section were only to clarify the algorithm but more simulations for this algorithm has to be done while taking into account a bigger number of transactions and data items.

An extension to this algorithm could be as well studying whether the flow of update messages can be performed in a dynamic way where one message can be sent containing updates of several data items instead of sending several messages per update of each data item in order to minimize traffic on the grid and this will depend on the transactions and the data items that are being accessed.

Naturally, however, further research and testing in real grid environment and relevant infrastructure should be done before grid technology and grid database are ready for use.
REFERENCES