Dynamic Data Replication in the Grid with Freshness and Correctness Guarantees

Inauguraldissertation

zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

Laura Cristiana Voicu
aus Rumänien

Zürich, 2011

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von:

Prof. Dr. Heiko Schuldt   Prof. Dr. Yuri Breitbart

Basel, den 8.12.2009

Prof. Dr. Eberhard Parlow
Dekan

Zusammenfassung

Diese Dissertation untersucht wichtige Aspekte von Daten-Grid-Infrastrukturen aus Architektur- und Performance-Sichtweise. Das Ziel ist eine skalierbare Infrastruktur für die dynamische Verwaltung replizierter Daten, die Korrektheit und verschiedene Frischgrade der Daten anbietet. Die Arbeit verfolgt dabei einen Ansatz, der auf einer verteilten Middleware basiert. Die Herausforderung dabei ist, eine Infrastruktur zu entwickeln, die sowohl Skalierbarkeit und Leistungsfähigkeit, als auch transaktionelle Garantien in sich vereint. Generelle Zielstellung dieser Dissertation ist es, ein neues Replikationsverfahren bereitzustellen, das sowohl dynamische Skalierbarkeit als auch Leistungsfähigkeit und globale Korrektheit unterstützt.

Wir betrachten dazu zunächst wichtige Aspekte der Grid-Infrastrukturen. Zudem zeigen wir Anwendungsfälle in den neu entstehenden eScience-Gebieten auf, für die ein dringender Bedarf nach integrierenden Replikationsverfahren besteht. Dazu stellen wir unsere Protokoll vor: Re:GRIDiT wird völlig neue Verfahren zur Replikation in dynamischen Grids ermöglichen, die auch den Zugriff auf konsistente Daten garantiert. Re:GRIDiT besteht aus drei verschiedenen Protokollen, die auf drei kritische Aspekte von Grid-Infrastrukturen abzielen.

Im Kontext komplexer Anwendungsanforderungen fokussieren wir uns auf den komplexeren und generelleren Fall mit verteilten Update-Transaktionen auf Replikaten. Um sowohl die Vorteile der synchronen Replikation als auch der asynchronen Replikation zu nutzen, ist eine Kombination der jeweiligen Verfahren nötig. Dazu entwickeln wir zunächst ein Protokoll für die verteilte Verwaltung von konkurrierenden Updates auf replizierten Daten, das Zugriffe auf Replikate kontrolliert und die Konsistenz der Replikate gewährleistet. Re:SYNCiT unterstützt die synchrone Datenreplikation, bei der zu jedem Zeitpunkt nur ein einziger konsistenter Zustand eines Datenobjekts sichtbar ist. Die Replikation von Daten ist also transparent für die Benutzer. Darüber hinaus berücksichtigt Re:SYNCiT die speziellen Merkmale der Grid Datenstrukturen, wie zum Beispiel die Versionskontrolle oder die Unterscheidung zwischen unveränderlichen und veränderlichen Datenobjekten. Die Grundlage dazu bildet ein formales Gerüst, das auch einen Korrektheitsbeweis ermöglicht.

Als nächstes betrachten wir eine Methode zum dynamisch verteilten Replikationsmanagement in Daten-Grid-Infrastrukturen. Wir schlagen in Re:LOADiT effiziente Algorithmen vor, die die Replikate optimal verteilt, um die Belastung auszugleichen. Durch den Einsatz von dynamischer Replikation kann das System jederzeit auf geänderte Lastbedingungen und unterschiedlichen parallelen Benutzerzugriff reagieren und die Anzahl der benötigten Replikate anpassen.

In einem weiteren Schritt verfeinern wir dieses Replikationsverfahren dahingehend, dass zudem der Frischegrad der Daten variieren kann. Die Idee ist es, Benutzern zu erlauben, Datenfrische gegen schnellere Antwortzeiten einzutauschen. Zu diesem Zweck führen wir einen Frischegrad und eine Versionsnummer als neuen Quality-of-Service Parameter für Anfragen ein, ohne dabei die Konsistenz zu verlieren. Dies ermöglicht "Frische-basiertes" Routing, das die verschiedene Frischegrade der Knoten in der Baumstruktur verwendet. Anfragen, die frische Daten benötigen werden an solche Knoten weitergeleitet die solche Daten besitzen. Anfragen nach weniger frische Daten werden in der Baumstruktur nach unten weitergeleitet bis Knoten mit dem entsprechenden Frischgrad erreicht werden.

Anschliessend sind wir auch an den Leistungscharakteristiken der neu entwickelten Verfahren interessiert. Dazu haben wir die vorgestellte Re:GRIDiT Algorithmen prototypisch implementiert und in eine realistische Grid-Infrastruktur mit 96 Knoten eingesetzt. Wir präsentieren die Resultate einer umfangreichen Evaluierung mit simulierter Arbeitslast, die einen realistischen Anwendungsfall vorgespiegelt. Es zeigt sich, dass dynamische Replikationsverfahren sehr effizient mit wenig Ressourcen und innerhalb eines angemessenen Zeitraums die Anzahl der benötigten Replikate anpassen. Es zeigt sich, dass Query Routing, das die Anfrage mittels einer Baumstruktur weiterleitet, die Antwortzeiten deutlich beschleunigen kann. Weiterhin ermöglicht Frische-basiertes Routing Benutzern, effektiv Datenfrische gegen Anfragegeschwindigkeit einzutauschen, ohne dabei die Konsistenz zu verletzen.

Zusammenfassend werden in dieser Dissertation neue Verfahren zur korrekten dynamischen Synchronisierung des Updates, zur Replikationskontrolle und zum Frischegrad der Daten in einem Daten-Grid vorgestellt. Die Verfahren sind sowohl auf einer formalen Grundlage aufgebaut, als auch vollständig in einem lauffähigen Prototypen implementiert. Mit einer umfangreichen experimentellen und analytischen Evaluation wurde abschliessend die praktische Verwendbarkeit des Verfahrens gezeigt.

Abstract

This thesis explores architectural issues and performance aspects of data Grid infrastructures. The objective is to develop a scalable infrastructure that is capable to dynamically manage replicated data in the Grid while at the same time providing freshness and correctness guarantees. We propose a decentralized middleware which can be deployed on top of any Grid (or any distributed, heterogeneous) infrastructure. The difficulty is to ensure that such an infrastructure can offer scalability, performance and correctness. The overall goal of this thesis is to present a replication mechanism that combines scalability, global correctness and quality of service guarantees in a dynamic way.

In the beginning we introduce important aspects of Grid environments and several scenarios from newly emerging eScience applications. These use case scenarios urgently require new integrated approaches to dynamic replication in a data Grid. Our main contribution is the Re:GRIDiT protocols that dynamically manage replicas in the Grid, while at the same time providing freshness and correctness guarantees. The Re:GRIDiT family consists of three different protocols which target the three main problematic aspects identified in current data Grid infrastructures.

Inspired by the requirements deduced from these scenarios we first concentrate our efforts on the more complex and general case of distributed update transactions on replicated data. We devise a protocol for the correct synchronization of concurrent updates to different updateable replicas in order to ensure their subsequent propagation to read-only replicas in a completely distributed way. Re:SYNCiT hides the presence of replicas to the applications, takes into account the special characteristics of data in the Grid such as version support, distinction between mutable and immutable objects, and provides provably correct transactional execution guarantees without any global component.

The next step is the Re:LOADiT approach to dynamic distributed replica management in data Grid systems. We propose efficient algorithms for selecting optimal locations for placing the replicas so that the load among these replicas is balanced. Given the data usage from each user site and the maximum load of each replica, our algorithm efficiently manages the number of replicas required, reducing or increasing their number.

Until now our approach dictates how update sites behave and from a user's point of view the clients will always access the most up-to-date data. We further refine this approach and introduce the Re:FRESHiT protocol, which allows to effectively trade freshness for performance and addresses freshness and versioning issues, needed in many Grid application domains, without losing consistency. Queries with different freshness levels are cleverly routed along our tree strategy, by taking advantage of the tree structure.

Finally we are also interested in the performance characteristics of the presented algorithms. We have implemented the Re:GRIDiT protocols using state-of-the-art Web service technologies which allows an easy and seamless deployment in any Grid environment. The evaluation has been conducted on up to 48 update sites and 48 read-only sites. We have used simulated workloads that mimic the behavior expected from our use case applications. Our evaluations have shown that the proposed Re:GRIDiT protocols are efficient, as replicas are created and/or deleted on demand and with a reasonable amount of resources. Dynamic changes in the tree structure allow flexible and efficient query routing along the tree structure. Clever routing strategies ensure an increased performance for queries with different freshness levels. Re:GRIDiT ensures replica consistency and is capable of providing different degrees of consistency and update frequencies.

Summarizing, this thesis presents new approaches for the correct synchronization of updates in a dynamic manner, replication management, and freshness guarantees in a data Grid. These approaches are founded on formal theoretical background and implemented in a full-fledged prototype in a realistic Grid environment. These approaches have been proven to be scalable by means of an extensive analytical and experimental evaluation.

Acknowledgements

First and foremost, I would like to thank my advisor, Prof. Dr. Heiko Schuldt, for valuable discussions, advice, and the constant support and inspiration to continue my Ph.D. studies. Second, I want to thank my former adviser at UMIT/ETH Zurich, Prof. Dr. Hans-Jörg Schek for the opportunity to start a Ph.D. career and supporting me in the beginning of my PhD. I wish to thank my co-referee, Prof. Dr. Yuri Breitbart at University of Kent, Ohio, USA, for the valuable discussions that have led to the work presented in this thesis.

This thesis could not have been possible without financial support from different projects. I am thankful that the following projects have supported work done within this thesis:

- DILIGENT (a testbed DIgital Library Infrastructure on Grid Enabled Technology) - project funded by the European Union under contract No. IST-2003-004260.

- COSA (COmpiling Software Architectures) - project funded by the Hasler Foundation.

I am also grateful to all of my colleagues and former colleagues in the Department of Computer Science at University of Basel, for many helpful discussions and a pleasant working environment: Fuat Akal, Gert Brettlecker, Nadine Fröhlich, Christoph Langguth, Diego Milano, Thorsten Möller, Paola Ranaldi, Michael Springmann.

Basel, November 2009
Laura Voicu

Contents

1

Introduction

Current trends in scientific research indicate a shift toward multi-scale applications, involving multi-disciplinary teams, often geographically dispersed. Aiming to provide an infrastructure that can sustain these applications the Grid has appeared in the mid 1990s as a proposed distributed computing infrastructure for advanced science and engineering. The choice of the name "Grid" to describe this infrastructure resonates with the idea of a future in which computing resources, compute cycles and storage, as well as expensive scientific facilities and software, can be accessed on demand like the electric power utilities of today. Within the context of rapid technological changes and advances and evolving users requirements, the Grid is challenged with providing increased opportunities for scientific data resources sharing and collaboration, distributed computation and distributed data storage. But the Grid goes beyond sharing and distributing data and computing resources. For thousands of scientists around the world, the Grid offers new and more powerful ways of working, such as distributed computing for large-scale data analysis or collaborative work. Many of the Grid applications require large amounts of data and as a result research into cutting edge data management aspects will have applications in bioinformatics, engineering, and chemistry. The term eScience has appeared as a consequence and defines global collaboration in key areas of science, and the next generation of infrastructure that will enable it. Through its capabilities it will change the dynamic of the way science is undertaken. Examples of eScience applications that can profit in one way or the other from Grid capabilities range from Bioinformatics/Functional genomics or Collaborative Engineering to Medical/Health care informatics, Earth Observation Systems, Virtual Observatories, Robotic Telescopes or Particle Physics at the LHC.

1.1 Trends and Applications in Grid Computing

The dawn of a new age – The Computer Era – glows before us with the promise of new and improved ways of thinking, living and working. The amount of information in the world is said to be doubling every two to three years [107]. The only way to keep up with this rapid increase in the amount of data and information is access to computers and the ability to control them for a particular purpose. To address this, people have already created computer networks that enable computer resources to be shared with every other computer in the network, moving from peer-to-peer networks, to supercomputers, or to Grids. And suddenly, there is so much hype on Grid computing. But what exactly is the Grid? Is it really a new thing? Or is it quite simply a clichéd "old-wine in new bottle"?

The term "the Grid" was coined in the mid 1990s to denote a proposed distributed computing infrastructure for advanced science and engineering [112]. Considerable progress has since been made on the construction of such an infrastructure, but the term Grid has also been conflated, at least in popular perception, to embrace everything from advanced networking to artificial intelligence [82]. In recent years a new tendency has been observed, resulting in the movement of the Grid from the purely academic to the highly popular. The Grid integrates distributed computational and data resources to create a single virtual resource which provides potentially unlimited processing and on-demand data storage power. In contrast to the first Grid applications which were developed for physicists, the Grid no longer exclusively targets scientific applications working with mostly read-only data. The more the Grid evolved, the more functionality it acquired, moving from the simple computational Grid, whose sole purpose was to gather together computing power (i.e., CPU) from distributed computers all around the world, to a new form of Grid which encompasses not only data storage space, but also data access, applications and services which can perform various types of computations or data manipulation. Grids have typically been divided into three types, on the basis of their use [112]:

Computational Grids: These Grids provide secure access to a huge pool of shared processing power suitable for high throughput applications and computation intensive computing.

Data Grids: Data Grids provide an infrastructure to support data storage, data discovery, data handling, data publication, and data manipulation of large volumes of data actually stored in various heterogeneous databases and file systems.

Service (Utility) Grids: This is the ultimate form of the Grid, in which not only data and computation cycles are shared but software or just about any resource. The main services provided through utility grids are software and special equipment. For instance, the applications can be run on one machine and all the users can send their data to be processed to that machine and receive the result back.

We focus our efforts on the second type of Grids, namely data Grids, whose aim is to share and manage huge volumes of distributed data.

1.2 Trends in eScience Application Domains

The applications being developed on the Grid benefit from Grid technologies in different ways. For some it is a matter of being able to access and control remote resources – instruments, compute resources, visualization or data resources. For others it is a matter of being able to collaborate with remotely located colleagues or specialists. Indeed, in some cases the Grid has provided a mechanism for new methodologies of scientific investigation – the ability to combine real-time experimental data with simulation data and have a distributed team visualize the results; the ability to collect data by remote senses and integrate that into simulations or analyses in, for example, agricultural or environmental settings, or in a medical application.

Since Grid technologies have been developed for applications with large storage and computation requirements in mind, they offer a promising tool to deal with, for instance, current challenges in many medical domains involving complex anatomical and physiological modeling of organs from images or large image database assembling and analysis. Digital medical images [4] represent a tremendous amount of data. In industrialized countries, a hospital produces several Terabytes of medical image data each year, bringing the total production of the European Union or the USA for instance to thousands of Terabytes a year. These data need to be properly archived for both medical and legal reasons. Beyond the outstanding issue of proper storage and long term archiving of such an amount of data, automated analysis is increasingly needed as manual inspection of medical images is a complex task, and may become extremely tedious and error prone. In the new emerging field known as eHealth, long-term, long-scale epidemiological studies, as well as the every day needs of medical scientists are facing some major challenges, including:

- The highly distributed and heterogeneous nature of virological, immunological, clinical, and experimental data,

- The high dimensionality and complexity of the genetic and patient data,
- The inaccessibility and lack of interoperability of advanced modeling, simulation, and analysis tools as well as the lack of an efficient data replication protocol to support such complex automated analyses by offering:
    - Access efficiency (moving data near processing),
    - Load balancing (distributing access load),
    - Security (data protection, moving processing near data if data confidentiality is an issue),
    - Availability (off-line access),
    - Reliability (disaster recovery, avoiding single point of failure).

Recent advances in Grid computing tackle only some of these problems by virtualizing the resources (data, instruments, computing nodes, tools, and users) and making them transparently available. Nevertheless, whereas some key issues can be solved using today's Grid technologies, in most aspects Grid technologies are still in their youth and often propose only very generic services. Replication of data across the nodes of the Grid has to grow beyond the simple mechanism of duplicating files that it is at the moment. Replication management in the Grid should to be able to deal with a potentially large number of updateable replicas per data object and provide transparent and consistent access to distributed data. It should be able to dynamically control the management of replicas, by taking into account resource consumption and loads when accessing a replica. Last but not least an efficient replication management has to be able to trade accuracy for performance when accessing data in the Grid. Despite the considerable work done in the context of distributed transaction management and replication management, there is no protocol which can be seamlessly applied to a data Grid environment without impacting correctness and/or overall performance.

1.3 Contribution and Scope of the Thesis

Recent trends in the Grid evolution aim at establishing data Grids as environments that enable users to effectively manage, share and publish resources and to provide services to support scientific research, technological innovation or cooperative teamwork. There is an increasing need to adequately and effectively manage resources, and in this context replicated data management still remains a challenge.

Use case scenarios in newly emerged eScience domains identify the following problems in current Grid infrastructures. First, current Grid replication solutions [118, 169] lack support for multiple concurrent updates to several replicas in a consistent manner. Second, they should take into account the semantics of the data which are managed in the Grid: mutable data can be subject to updates; immutable data, in turn, cannot be changed once created, but may be subject to version control. To the best of our knowledge, none of the existing Grid replication mechanisms makes this distinction. Third, Grid replication solutions need to be able to support dynamic replica management and deployment (i.e., creation and deletion of replicas) in order to increase the performance and to scale in numbers and geographical area. Most existing solutions rely on (some) centralized components [70] or do not address scalability and high performance issues. Fourth, new solutions for data Grid replication need to be able to take into account user demands. Users may have different requirements regarding how "fresh" their data should be. In addition, since user queries are not known in advance and may not be compliant with current replica placement, replication management in the Grid needs to support heavy load for geographically distributed queries.

Driven by the need and opportunity to bring Grid capabilities to these application domains, we envisage a Grid infrastructure that evolved from a tool to solve computational and data-intensive problems towards a general-purpose infrastructure with complex, heterogeneous and dynamic requirements. Availability, dependability and scalability are an issue in today's Grids and our goal is to solve these problems through a protocol that provides reliable and efficient access to distributed and heterogeneous data anytime, anywhere, and the capability to conduct long-term, large-scale statistical studies.

This thesis investigates architectural issues and performance aspects of data Grid applications. The objective is to build a basic infrastructure capable of supporting these applications by means of complex algorithms for the globally correct distributed synchronization of updates, dynamic load balancing for replica selection and deployment and freshness-aware scheduling and routing of queries that allow a trade-off between freshness of data and query performance.

The main contribution of this thesis is Re:GRIDiT (Replication Management in Data GRid Infrastructures using Distributed Transactions), a scalable infrastructure that builds upon several new algorithms for the dynamic synchronization replicas in a data Grid in a distributed way and seamless support for read-only requests with different freshness levels. More concretely, Re:GRIDiT consists of three different protocols which target the three main problematic aspects identified in current data Grid infrastructures. The three protocols are:

Re:SYNCiT provides new protocols for the correct synchronization of concurrent updates to different updateable replicas and a system model able to handle the different semantics of data in a seamless way, completely transparent to the transactions and the users. The results of the experimental evaluation of the Re:SYNCiT protocol for update transactions have shown both performance and scalability when applied at Grid scale.

Re:LOADiT is dynamic in a way that according to a combination of current load, host proximity and freshness criteria, new replicas can be created or removed on demand. The evaluation of the Re:LOADiT protocol has shown how load metrics can be used to increase the throughput by dynamically (un)deploying replicas with a reasonable amount of time and effort.

Re:FRESHiT provides read-only transactions the full flexibility to specify the freshness (for mutable data) or version number (for immutable data). The Re:FRESHiT protocol is capable of supporting read-only requests with different freshness levels and to route queries in an efficient way, especially when it comes to trading accuracy for performance while accessing data in the Grid.

Finally, as an integral part of this work, we present an experimental evaluation of the proposed algorithms using an eScience use case scenario as input. We have developed a prototype system which is used to evaluate the performance and scalability of our approach under the influence of a variety of parameters. Our evaluations show that our optimistic concurrency control protocol for update replicas outperforms traditional pessimistic approaches for low to medium conflict rates. Dynamic changes in the load determines when new replicas need to be acquired or released, with minimum amount of resources. Furthermore, clever refresh and routing strategies show an almost 20% to 30% improvement in the query response time for user queries with lower freshness levels. The proposed algorithms have been evaluated independently in a realistic Grid environment of up to 96 sites.

1.4 Structure of the Thesis

This thesis is organized as follows. Chapter 2 presents an overview of application areas in the eScience domain of relevance for data Grid replication and how these use cases have influenced the main features of the Re:GRIDiT system. This chapter presents an earth observation application scenario used throughout the thesis. In Chapter 3, we discuss foundational aspects of the theory of transaction and replication management and survey related work in the field. At the same we explain how existing models and approaches are unable to deal with the problems imposed by current data Grid applications. In Chapter 4, we describe the data Grid replication infrastructure and the basic components needed for this purpose. The model we propose is based on assumptions that are derived from the analysis of application specific requirements. In order to address the particular needs of the data Grid applications we have identified three stages for our dynamic, freshness aware replication management protocol. In Chapter 5 we solve the more complex problem of synchronizing updates to several replicas in a grid in a completely distributed way by introducing the Re:SYNCiT protocol. Chapter 6 explains how the Re:LOADiT protocol for dynamic replication is capable to support on-demand replica deployment and placement. In Chapter 7 we further enhance our system with the ability to support read-only queries with different freshness levels in the Re:FRESHiT protocol. Chapter 8 describes the implementation of the Re:GRIDiT system in a data Grid infrastructure. Technical details on the implementation of the Re:GRIDiT system of are presented. Chapter 9 empirically proves the applicability and performance of the presented infrastructure and the proposed protocols for dynamic replica management and freshness aware query routing through evaluations within the infrastructure implementation of Re:GRIDiT. Furthermore, in Chapter 10 we explained how our protocol can be seamlessly applied to a Data Cloud environment. Finally, Chapter 11 concludes by summarizing the impact of the presented work and discusses open and future research issues.

2 Motivation

In this chapter we motivate the applicability of our approach to dynamic management of replicated data in the Grid with freshness and correctness guarantees to various application domains. In particular, we introduce applications in the eScience domain of relevance for data Grid replication. In order to get a more precise view of the application, we present a motivating earth observation application scenario in detail which will be used throughout the remainder of the thesis for motivation and illustration purposes. We use this example to motivate the choice of our system model for data Grid replication presented in Chapter 4 and the building blocks of our approach, which will be introduced in Chapters 5, 6 and 7.

2.1 Example Earth Observation Scenario

In the Earth Observation domain, earth observation data are acquired from satellites, sensors and other data acquisition instruments, archived along with metadata, catalogued and validated. According to [102], by the year 2010 the earth observation data archives around the world is estimated to grow to around 9.000 Terabytes and by the year 2014 to around 14.000 Terabytes. Beyond the outstanding issue of proper storage and long term archiving of such an amount of data, automated analysis is increasingly needed as manual inspection of images is a complex and error prone task, and becomes infeasible for practical applications. Furthermore, in crucial decision-making situations, researchers need not only aggregate data and computing power to support such decisions, but also human expertise.

Consider for example the following earth observation scenario (see Figure 2.1), where data are collected at one or more stations and maintained

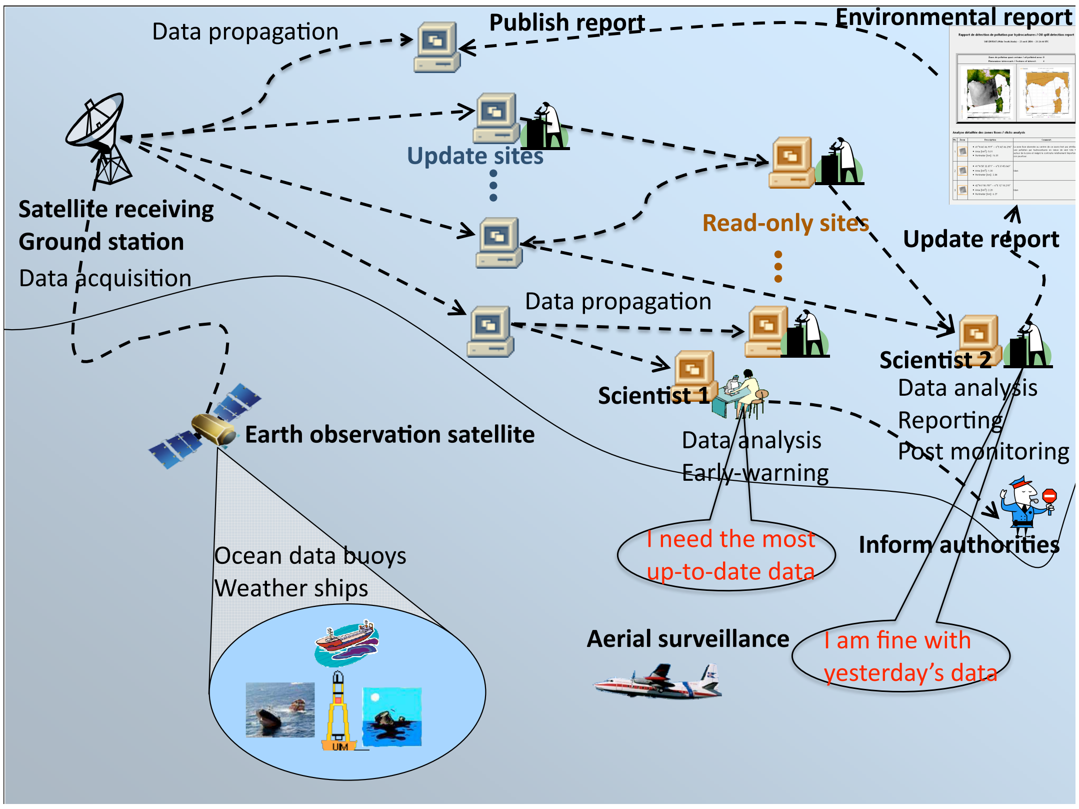


Figure 2.1: Example Earth Observation Application Scenario

at geographically distributed sites. At the same time, environmental reports that include satellite images of the region observed, their descriptions (in the form of XML documents) and image interpretations [40] are periodically generated and/or updated at several sites. In order to improve availability, copies of data are maintained at different sites.

In our example, Scientist 1 is closely monitoring oil spills in the sea. Spillage of oil in coastal waters is one of the most hazardous events to occur. The potential damage to the natural and economic health of the area at stake requires the readiness to detect, monitor and clean-up any large spill in a rapid manner. Satellite data acquisition and data distribution among surveillance sites can contribute to early warning and near real-time monitoring. Consider for example the case of Scientist 1 who demands the most up-to-date data, for this real-time monitoring. She will integrate information from in situ, airborne and space-based observations and apply data assimilation models in order to support early detection of outliers which correspond to significant and critical events. This type of study is particularly important for damage prevention but also for risk management in order to prepare existing infrastructure inventories in high risk areas and to define and structure institutions and resources within the region in terms of their involvement and role in the damage reduction. The most up-to-date data for this type of applications, even in the presence of potentially many concurrent updates, are a fundamental requirement that allows to deal effectively with risk reduction activities and with scheduling of the human resources required to carry out response activities, to utilize, in the most efficient manner possible, existing facilities and resources, to avoid duplication and to optimize the use of limited resources. At the same time, other scientists at different locations will consider additional data, not available to the first scientist, and consequently update existing reports. Since they are working with different replicas of the same original reports, their updates will be sent to the originating replicas first but will need to be synchronized with all other replicas of the reports in order to guarantee consistency. Moreover, this type of applications usually involves a broad range of user communities, including managers and policy makers in the targeted societal benefit areas, scientific researchers and engineers, civil society, governmental and non-governmental organizations and international bodies, working together to analyze the same data for monitoring, predicting, risk assessment, early warning, mitigating, and responding to hazards at local, national, regional, and global levels. According to [73], the European Space Agency alone currently has several thousands of registered data users, and their number is continuously increasing.

This scenario confirms the statement that data Grids no longer target applications that need read-only data; updates occur and they need to be properly synchronized among replicas. We address this issue in the Re:SYNCiT protocol.

Replication techniques are able to reduce response times by moving and distributing content closer to the end user, speeding content searches, and reducing communication overhead. An efficient replication management protocol has to consider changes in the access patterns, i.e., it should dynamically provide more replicas for data objects which are frequently accessed. Consider the unfortunate case when an accident has occurred. The urgency of the situation requires more reports to be generated and increasing requests to access particular data objects, relevant to the region where the accident has occurred. Therefore the number of updateable replicas for data objects of importance should be dynamically increased so that the system is capable to serve requests in a timely manner. However, increasing the number of updateable replicas per data object in an unlimited way may have significant drawbacks on the overall system performance. Therefore, the number of updateable replicas for data objects which are no longer of importance to a large group of users should be dynamically reduced, in order to reduce the overhead of replica maintenance.

Optimizing access cost of data requests and reducing the cost of replication are two conflicting goals and finding a good balance between them is a challenging task. This task is successfully fulfilled by the Re:LOADiT protocol.

Consider another user, Scientist 2, who is performing oil slick distribution studies, in order to determine the environmental impacts of stranded oil, and offer recommendations of cleanup procedures and methods least likely to exacerbate the effects caused by the oil. In order to perform a thorough investigation he will require several successive data acquisitions (before and after the oil spill), combined with wind, sea-state and other meteorological data for complementing the information sources. He requires therefore older versions for his studies, from multiple archives of satellite images, and previous reports of other scientists. A possible course of actions can be: he acquires first radar imagery of oil spills and decides to retrieve a complementary optical imagery and overlay it to the initial one. He overlays then the resulting imagery with tracks of major tanker routes to highlight any correlation and checks it against a map showing coastlines of maximum biodiversity or a mosaic of chlorophyll distribution. Finally, he applies wave and wind meteorological information layers to model the behavior and impact of these spills. The results of his analysis will be used to update existing environmental reports, and will include new and old time series, re-generated maps together with his own interpretations of the data. Over longer time scales, this type of damage assessments provides basis for monitoring and recovery assessment programs. Data management in the Grid needs to take into account the requirement of both scientists: (i) to keep large volumes of data, (ii.) to update (parts) of the data as new findings come out, (iii.) to access the most recent version of all data items, and (iv.) to also keep outdated versions which have been updated in the meanwhile, for read access.

For the latter, the notion of data freshness needs to be supported in order to specify the staleness of data and to allow users to specify how old the data they ask for may be. Re:FRESHiT efficiently uses freshness criteria to make replica selection and query routing decisions.

2.2 Example eHealth Scenario

The availability of digital images inside hospitals and their ever growing inspection capabilities have established digital medical images as a key component of many pathologies diagnosis, follow-up and treatment. To face the growing image analysis requirements, automated medical image processing algorithms have been developed over the two past decades. In parallel, medical image databases have been established in health centers. Grid technologies appear to be a promising tool to face the raising challenges of computational medicine. They offer wide area access to distributed databases in a secure environment and bring the computational power needed to complete large-scale, long-term statistical studies.

Multiple sclerosis, for instance, is a severe brain disease that affects about 0.001% of the population in industrialized countries and for which no complete redemption treatment exists. Currently few drugs are available on the market that can slow down the brain impairment caused by the disease, and unfortunately their efficiency is difficult to quantitatively assess and their real effect is rather controversial. Assessments of these therapies have been proposed through serial Magnetic Resonance (MR) images of the head by measuring the brain white and gray matter atrophy resulting from the disease [58, 122]. However, this parameter extraction requires complex image analysis algorithms since very small volume variations are significant (the normal brain atrophy due to aging is in the order of 0.5% per year, while the disease may lead to an accelerated atrophy in the order of 1% per year). Therefore, only studies involving a large number of patients over a long period of time prove to have a statistical significance. Such an epidemiological study involves at least hundreds of patients (a group of placebo patients and several groups of treated patients following an experimental protocol) over years (an MR acquisition every few months is required to build time series). This kind of clinical protocol results in the acquisition of thousands of images, 10 to 20 MB each, summing up to Terabytes of data.

At the same time, eHealth systems should focus on prevention and early diagnosis as well as treatment; they should enable self-management of diseases and care at homes by the individuals or their families. Such proactive personal health systems have the potential to improve public health and significantly lower the health care costs. The wireless medical sensors, digital home technologies, cognitive assistance, advanced robotics for care support, context aware applications and services, and intelligent proactive computing technologies are the enabling technologies of this vision, but at the same time continuously generating huge amounts of data (in the form of monitoring videos and continuous data streams).

While medical images are stored once and never updated, the situation for medical records is different. Data originating from physiological sensors need to be aggregated and medical histories of patients need to be frequently updated. Similarly, medical reports which include interpretations of physiological data and/or medical images need to be updated, for instance, by appending new diagnoses for newly created medical images or for new data created by physiological sensors.

Moreover, different medical scientists may have different requirements regarding how up-to-date their data should be. Consider patient X, suffering from stable cardiovascular disease and traveling in Europe. In case of an emergency, medical scientist W in the visiting country needs access to his most up-to-date medical records to ensure that he receives adequate treatment. Consider now medical scientist M who would like to identify patients that have similar pathological deviations in the X-ray of their lung as patient Z, for whom SARS has been diagnosed. For the purpose of this epidemiological study across a set of patients, he is satisfied with last week's data. His analysis reports will be published in the Grid to be made available to hundreds of scientists working in the same field and sharing the same data [57]. Together with robust mechanisms and policies needed to ensure that patient privacy and confidentiality are preserved, the delivery of such repositories of medically rich information for the purposes of scientific research is urgently required.

As in the previous earth observation scenario, eHealth applications in the Grid need more flexible ways of managing resources in order to dynamically distribute data according to the access characteristics of their users. In addition, to overcome the limitations of synchronous replication while at the same time to better meet user requirements, data should be made available with different levels of freshness.

2.3 Example Storm Modelling Scenario

Meteorologists and environmental modelers have been attempting to build a Grid to enable them to accurately predict the exact location of severe storms such as tornadoes, based on a combination of real-time wide-area weather instrumentation and large-scale simulation coupled with data modeling [79]. This is an extremely difficult problem and so far beyond the current capabilities of storm simulation.

Suppose a researcher wishes to understand why some severe thunderstorms produce a succession of multiple tornadoes, while others do not. The first step requires establishing a climatology of observed storm behavior for comparison with numerical simulations. He/She would need to locate, access, and decode all required data – including ten years of Doppler radar data, along with upper air observations and model forecasts, hourly surface observations, weekly land surface data, 6-minute precipitable water data from GPS satellites, and 15 minute satellite radiance data – all for the contiguous region observed. The researcher then accesses the appropriate subset of data, which are too voluminous to be stored locally and must be stored on a remote site.

Using feature detection and pattern recognition techniques, the researcher applies a data mining engine to the assimilated data sets to catalog all cyclic versus non-cyclic storms, the existence of tornadoes, and the surrounding environmental conditions associated with each. The resulting metadata, along with the assimilated data sets, can then be made available for use by the broader community, even though the raw data physically reside elsewhere.

The researcher then develops numerical simulations designed to provide an understanding of the storm cycling process. The simulations produce hundreds of Terabytes of output, and mining techniques are used to correlate cyclic storm behavior with environmental characteristics and internal storm dynamics. The simulation output are automatically published to geographically distributed digital library catalogs. The mining tools trigger the ensemble system over appropriate domains, which in turn automatically requests Grid computing resources with sufficient priority to provide results significantly faster than the weather unfolds (so-called better than real time predictions). This on-demand requirement for additional resources should be handled automatically by the Grid.

In this scenario we notice again several aspects that have motivated our Re:GRIDiT approach, namely the highly dynamic nature of the application and the need to support a complex data model.

2.4 Summary

The Grid started as a vision to share potentially unlimited computer power and data storage capacity over the Internet. It made big steps towards becoming highly popular by making contributions to scientific research, helping scientists around the world to analyze and store massive amounts of data. But in contrast to the first Grid applications which were developed for physicists, the Grid no longer exclusively targets scientific applications working with mostly read-only data. The more the Grid evolved, the more functionality it acquired, moving from the simple computational Grid, whose sole purpose was to gather together computing power (i.e., CPU) from distributed computers all around the world, to a new form of Grid which encompasses not only computing power and storage space, but also data, applications and services which can perform various types of computations or complex data manipulations.

As a consequence new eScience domains have emerged with the purpose of enabling the cooperation of distributed research groups who share data and powerful computing environments. Immense data sets that are produced by expensive equipment need to be accessed and evaluated by collaborating research groups who are working at distant locations. An efficient data replication protocol can guarantee the success of such large-scale collaborative studies, by ensuring, to mention only a few of the advantages: availability of data, efficient access to data (using dynamic load balancing mechanisms) and reliable and consistent access to data, while at the same time answering the day-to-day needs of scientists and researchers, which as potential data Grid users, may have different requirements from a data Grid environment, in particular regarding how "fresh" their data should be. Motivated by these application scenarios and having in mind the above mentioned fundamental requirements we have developed the Re:GRIDiT approach to dynamic replication in data Grid environments with freshness and correctness guarantees. The Re:GRIDiT family consists of three core pillars which target the three main problematic aspects identified in current data Grid infrastructures, and which will be presented in details in Chapters 5, 6 and 7.

3

Foundations Of Transaction and Replication Management

This chapter concentrates on the foundations of transaction and replication management. Transaction management ensures that concurrent operations, be they queries or updates, are correctly executed. Replication management guarantees that changes made to data are eventually propagated to all copies of the data. Section 3.1 presents a classification of transaction and concurrency control approaches relevant to this thesis. Section 3.2 presents a survey of existing replication management approaches and their characteristics. After establishing the basis for the discussion of the correct execution of distributed, concurrent operations on replicated data, we discuss how current approaches fail to meet our requirements and cannot be applied to Grid environments. Consequently, in Chapters 5, 6 and 7 we present a new approach for the dynamic management of replicated data in the Grid with freshness and correctness guarantees, which is a careful combination and extension of different scheduling and replication control techniques as discussed in the following.

3.1 Transaction Management Theory

Transaction management deals with the problems of keeping the database in a consistent state even when concurrent accesses and failures occur [133].

A transaction consists of a series of operations performed on one or several databases. An important aspect of transaction management is that if a database was in a consistent state prior to the initiation of a transaction, then the database should return to a consistent state after the transaction is completed. This should be done irrespective of the fact that transactions were successfully executed in a concurrent manner or there were failures during the execution [133]. Thus, a transaction is a unit of consistency and reliability. The properties of transactions are discussed later in this section.

A transaction either succeeds or fails as a unit. A transaction can be terminated in two ways: committed or aborted (cancelled). When a transaction is committed, all changes made within it are made durable (forced on to stable storage). When a transaction is aborted, all changes made during the lifetime of the transaction are undone.

Traditional transactions are typically referred to as ACID transactions [99]. A transaction has four properties that lead to the consistency and reliability of a database, called ACID properties:

Atomicity: A transaction's state changes are atomic: they either all happen or none happen. In other words, the transaction completes successfully (commits), or if it fails (aborts), all of its effects are undone.

Consistency: A transaction is a correct transformation of state. The actions taken as a group do not violate any of the integrity constraints associated with the state. Transactions produce consistent results and preserve application-specific invariants.

Isolation: Intermediate states produced while a transaction is executing are not visible to other transactions. Furthermore transactions appear to execute serially, even if they are actually executed concurrently.

Durability: The effects of a committed transaction are never lost (except by a catastrophic failure).

The most important aspects of transaction management are concurrency control to guarantee the isolation properties of transactions, for both committed and aborted transactions and recovery to guarantee the atomicity and durability of transactions.

Concurrency control is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other. As such, it controls the interleaving of concurrent transactions, to give the illusion that transactions execute serially, one after the other, with no interleaving at all. Interleaved executions whose effects are the same as serial executions are called serializable.

Recovery is the activity of ensuring that software and hardware failures do not corrupt persistent data. A recovery algorithm monitors and controls the execution of programs so that the database includes only the results of transactions that run to a normal completion. As such, it must ensure that the results of transactions that do execute are never lost. Moreover, if a failure occurs while a transaction is executing, and the transaction is unable to finish executing, then the recovery algorithm must wipe out the effects of the partially completed transaction. That is, it must ensure that the database does not reflect the results of such transactions.

We begin by discussing essential transaction models [192]: *flat transactions*, *chained transactions* and their generalization *layered transactions*. We discuss techniques for centralized and distributed systems, and for single copy, multiversion, and replicated databases. For this purpose we introduce different types of concurrency control algorithms and recovery mechanisms and the unified theory of concurrency control and recovery. This discussion aims to provide an exemplification of the fundamental properties of transaction models and the notion of correctness.

3.1.1 Flat Transactions

Conventional flat transactions are the simplest type of transactions and represent the basic building blocks for organizing an application into atomic actions (an all or nothing operation).

A flat transaction t is a partial order of basic operations which ends in either a $commit$ operation or an $abort$ operation. In the *read/write model*, the operations are of the form $r(x)$ ($read$) and $w(x)$ ($write$), where $x \in D$ is a data element of the database D. Formally:

Definition 3.1 ***(Flat Transaction)*** *A transaction is a tuple with* $T = (O, \ll)$ *consisting of a set of operations* O*, on which we define a (partial) order relation* $\ll$*, where the following relations hold:*

1. $O = \{op_1, op_2, ..., op_n\} \cup term$*, is a finite set of operations* $\{op_1, op_2, ..., op_n\}$*, and* $term \in \{commit, abort\}$ *a terminating operation (i.e., the last operation in a transaction according to* $\ll$*),*

2. $\ll \subseteq (O \times O)$ *is the precedence relation.*

□

The precedence relation $\ll$ establishes the execution order of the operations of a transaction and can be one of the two following types:

partial $\ll$: allows parallelism to take place within a transaction

total $\ll$: allows sequential execution of the operations of a transaction:
$T = \langle op_1 \ll op_2 \ll ... \ll op_n \ll commit \rangle$.

When transactions are executed concurrently, their operations are interleaved resulting into a so-called *schedule*.

Definition 3.2 ***(Schedule)*** *Let* $\tau = \{T_1, T_2, ..., T_n\}$ *be a (finite) set of transactions. A schedule* S *is a triple* $S = (\tau, \mathbb{O}, \ll_s)$. S *contains the execution order of all operations of the transaction in* τ*, for which the following relations must hold:*

1. $\mathbb{O} = \cup O_i$ *is the set of all operations of all transactions in* τ
2. $\ll_s \subseteq (\mathbb{O} \times \mathbb{O})$ *is a partial order, for which the following holds:* $\ll_i \subseteq \ll_s$ *for all* $T_i \in \tau$*, i.e., the scheduling order must respect the all transaction orders in the sense that all transaction orders are contained.*

□

Moreover, a schedule contains at most one commit or one abort for each transaction.

Transaction management has to ensure the ACID properties of all transactions within a schedule, even if transactions are executed interleaved. However, not all schedules lead to a correct execution of all the transactions that belong to the schedule. In order to guarantee correctness the notion of *serializability* is used. The basic idea is that a schedule S is correct if it leads to the same database state as the serial execution of the transactions in the transaction set. In other words, serializability ensures that there is no cyclic flow of information between transactions. A schedule S is considered correct if it equivalent to any serial execution, called *serial schedule*.

Definition 3.3 ***(Serial Schedule)*** *In a serial schedule the ordering of operations is total and for any* $i, k \in \{1, ..., n\}$ *the following holds: all operations of* T_i *are executed before any operation of* T_k. □

Two serial schedules that appear to be correct from a user's perspective can lead to different database intermediate and/or final states. The following notion of a *committed projection* helps to formally ignore operations of aborted transactions in a schedule [28]:

Definition 3.4 ***(Committed Projection)*** *For a given schedule* S*, the committed projection* $C(S)$ *of* S *is obtained by deleting all operations that do not belong to transactions committed in* S*, i.e., it is reduced to:* $\cup T_i : C_i$ *in* S. □

In other words, $C(S)$ does not contain either active or aborted transactions.

Serializability is a property of a transaction schedule (history). It relates to the isolation property of a database transaction. Serializability of a schedule means equivalence (in the outcome, the database state, data values) to a serial schedule (i.e., sequential with no transaction overlap in time) with the same transactions. It is the major criterion for the correctness of concurrent transactions' schedule, and thus supported in all general purpose database systems. The rationale behind serializability is the following: if each transaction is correct by itself, i.e., it meets certain integrity conditions, then a schedule that comprises any serial execution of these transactions is correct (its transactions still meet their conditions). "Serial" means that transactions do not overlap in time and cannot interfere with each other, i.e, complete isolation between each other exists. Any order of the transactions is legitimate, if no dependencies among them exists, which is assumed. As a result, a schedule that comprises any execution (not necessarily serial) that is equivalent (in its outcome) to any serial execution of these transactions, is correct.

Three major types of serializability exist [192]: final state serializability, view serializability, and conflict serializability. Final state serializability is the general definition of serializability. View serializability is a restriction of the final state serializability. Conflict serializability is a broad special case, i.e., any schedule that is conflict-serializable is also view-serializable, but not necessarily the opposite. Conflict serializability is widely utilized because it is easier to determine and covers a substantial portion of the view-serializable schedules.

Final state serializability. This is the most intuitive definition of serializability. Under this definition, two schedules are considered to be equivalent if they contain the same operations and have the same final effect on the state of a database [192].

View serializability of a schedule is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that respective transactions in the two schedules read and write the same data values ("view" the same data values) [192].

Conflict serializability is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically-ordered pairs of conflicting operations (same precedence relations of respective conflicting operations) [192].

Determining final state and view serializability of a schedule is an NP-complete problem [140, 139], and therefore this notion is difficult to use in practice. Conflict serializability can be determined in time polynomial in the number of transactions in the schedule. In the remaining of the thesis, we will use therefore the conflict serializability as correctness criterion.

Conflict serializability is based on the basic observation that the invocation order of some operations can be exchanged within a schedule without producing any changes (from the point of view of the operation and also from the point of the data objects). This observation holds not only for the read/write model but can also be generalized to any set of operations. Operations whose invocation order cannot be exchanged without producing any changes are said to be *conflicting*. In the read/write model, operations upon data are read or write (where a write is either insert or modify or delete).

Definition 3.5 ***(Conflict Relation)*** *Two operations* op_1, op_2 *are in conflict, if they are of different transactions, act upon the same data element, and at least one of them is a* write. *Each such pair of conflicting operations has a conflict type: It is either a read-write, or write-read, or a write-write conflict. The transaction of the second operation in the pair is said to be in conflict relation with the transaction of the first operation, i.e.,* $op_1 \text{ CON } op_2$. □

A more general definition of conflicting operations (also for complex operations, which may consist each of several basic (primitive) read/write operations) requires that they are non-commutative (changing their order also changes their combined result). Each such operation needs to be atomic by itself in order to be considered an operation for a commutativity check. Section 3.1.2 will detail this case.

The dependency relation of a schedule S contains all the conflict pairs of all transactions in S.

Definition 3.6 ***(Dependency and Dependency Relation)*** *Let* S *be a schedule. Operation* o_k *depends on operations* o_i *in* S*, shortly written as* $o_i \rightarrow o_k$ *if and only if:*

1. $i \neq k$, *i.e.,* T_i *and* T_k *are different transactions,*
2. $o_i \ll_s o_k$, *i.e.,* o_i *comes before* o_k *in* S,
3. $o_i \text{ CON } o_k$, *i.e.,* o_i *and* o_k *are in conflict,*
4. $C_i, C_k \in S$, *i.e., both transactions have finished.*

The dependency relation $\mathrm{dep}(S)$ *is the set of all dependent pairs in S:* $\mathrm{dep}(S) = \{(o_i, o_k) \in S \mid o_i \rightarrow o_k\}$. □

Schedule compliance with conflict serializability can be tested with the serialization graph (precedence graph, conflict graph) for committed transactions of the schedule [139]. It is the directed graph representing precedence of transactions in the schedule, as reflected by precedence of conflicting operations in the transactions.

Definition 3.7 ***(Serialization Graph)*** *The serialization graph* $SG(S)$ *of a schedule* S *is a graph where transaction are nodes and the edges are the pairs* T_i, T_k*, for which the following holds: there exist operations* $o_i \in T_i$ *and* $o_k \in T_k$ *with* $(o_i, o_k) \in dep(S)$. □

In other words, in the serialization graph transactions are nodes and dependency relations are directed edges. There exists an edge from a first transaction to a second transaction, if the second transaction is in conflict with the first, and the conflict is materialized, i.e., if the requested conflicting operation is actually executed. In many cases a requested/issued conflicting operation by a transaction is delayed and even never executed, typically by a lock on the operation's object, held by another transaction. As long as a requested/issued conflicting operation is not executed, the conflict is non-materialized. Non-materialized conflicts are not represented by an edge in the serialization graph.

The following observation is a key characterization of conflict serializability [74]:

Theorem 3.8 ***(Serializability)*** *A schedule is conflict-serializable if and only if its committed projection is acyclic.* □

This means that a cycle consisting of committed transactions only is generated in the serialization graph, if and only if conflict-serializability is violated. Cycles of committed transactions can be prevented by aborting an active (neither committed, nor aborted) transaction on each cycle in the serialization graph of all the transactions, which can otherwise turn into a cycle of committed transactions. One transaction aborted per cycle is both required and sufficient number to break and eliminate the cycle (more aborts are possible, and can happen in some mechanisms, but unnecessary for serializability). Such a situation is carefully handled, typically with a considerable overhead, since correctness is involved. Transactions aborted due to serializability violation prevention are restarted and executed again immediately. Serializability enforcing mechanisms typically do not maintain a serialization graph as a data structure, but rather prevent or break cycles implicitly.

In practical scenarios, failures of many types can compromise the atomicity and persistence of transactions, for example programming failures, power outages, human operator failures, hardware or software failure (which might imply the loss of permanently stored data), catastrophes. In order to still ensure correctness in the presence of failure, several strategies have been devised. Transactions that remained unfinished due to application failures need to be rolled back. In case of hardware or software failures, for example, the database management system needs to be restarted and crash-recovery protocols need to performed in which unfinished transactions need to be rolled back and finished transactions need to be restarted.

In general, when discussing concurrency control, failures are not taken into account, the implicit assumption being that the influence of an aborted transaction on the correctness of a schedule need not be considered. This assumption is, however, false, since write (update) operations of an aborted transaction could have changed the state of a database (and these changes could have already been read by other transactions).

A major characteristic of a database transaction is atomicity, which means that it either commits, i.e., all its operations' results take effect in the database, or aborts (rolled back), all its operations' results do not have any effect on the database ("all or nothing" semantics of a transaction). In all real systems transactions can abort for many reasons, and serializability by itself is not sufficient for correctness. Schedules also need to possess the recoverability property. Recoverability means that committed transactions have not read data written by aborted transactions (whose effects do not exist in the resulting database states). While serializability may be compromised on purpose in many applications for better performance, compromising recoverability would violate integrity of the database, as well as that of transactions' results. In some applications, absolute correctness is not needed. Commercial databases provide concurrency control with a whole range of isolation levels which are in fact (controlled) serializability violations in order to achieve higher performance. Higher performance means better transaction execution rate and shorter average transaction response time (transaction duration). Snapshot isolation [23, 76] is an example of a widely utilized relaxed serializability method. Note that the recoverability property is needed even if no database failure occurs and no database recovery from failure is needed, as it is needed to correctly automatically handle aborts, which may be unrelated to database failure and recovery from failure.

A transaction T_i reads data item x from T_k if T_k was the transaction that had last written into x but had not aborted at the time T_i read x. More precisely, we say that:

Definition 3.9 ***(Reads-From-Relation)*** *Let* r_i *be a read operation of transaction* T_i *and* w_k *a write operation of transaction* T_k. $r_i(x)$ reads-from $w_k(x)$, $i \neq k$, *if:*

1. $w_k(x) < r_i(x)$
2. *there exists no* $w_m(x)$ *such that* $w_k(x) < w_m(x) < r_i(x)$
3. T_k *is not aborted*

The Reads-From-Relation is defined as $\mathsf{RF} = \{(T_k, x, T_i) \mid r_i(x) \text{ reads-from } w_k(x)\}$
□

Definition 3.10 ***(Recoverability)*** *A schedule* S *is recoverable, if for all transactions* $T_i, T_k \in S$*, the following holds: If* $(T_k, x, T_i) \in \mathsf{RF}(S)$ *and* $i \neq k$ *and* $C_i \in S$*, then the commits must be ordered as follows:* $C_k < C_i$. □

A schedule avoids cascading aborts if transactions read only the changes of committed transactions (i.e., a transaction does not read an item changed by another transaction until that transaction has committed). Cascading aborts avoidance is sufficient but not necessary for a schedule to be recoverable.

A schedule is called strict if every value written by a transaction T is not read or changed by other transactions until T either aborts or commits.

Definition 3.11 ***(Strictness)*** *A schedule* S *is strict, if for all transactions* $T_i, T_k \in S$*, the following holds: If a write operation of* T_i*,* w_i*, precedes a conflicting operation of* T_k *(either read or write),* $w_i < o_k$ *then the commit event of* T_i *also precedes that conflicting operation of* T_k *:* $C_i < C_k$. □

Strict schedules avoid cascading aborts and allow efficient recovery of databases.

Concurrency Control Protocols

A concurrency control protocol is used to produce a serializable schedule for multiple concurrent transactions. Concurrency control techniques are of two major types:

Optimistic approaches aim at solving the problems after they occurred [114, 162]. The test for serializability is performed before the commit of a transaction. If a violation has occurred, the transaction is typically aborted. Otherwise it is committed.

Pessimistic approaches intend to avoid problems rather than solving them. Potential conflicts are detected in advance and a transaction blocks data access operations of other transactions when such potential conflicts are detected. This requirement ensures that operations that may violate serializability (and in practice also recoverability) do not occur. Typical representatives are two-phase locking protocols [74] or timestamp-based concurrency protocols [28, 155].

The main difference between the two types of approaches is the way conflicts are handled. A pessimistic method blocks a transaction operation upon conflict, and generates a non-materialized conflict, while an optimistic method does not block, and generates a materialized conflict. At any method conflicts are generated by the way transaction operations are scheduled, independently of the method. A cycle of (materialized) conflicts in the serialization graph (conflict graph) represents a serializability violation, and should be avoided or eliminated in order to maintain serializability. A cycle of (non-materialized) conflicts in the wait-for graph represents a deadlock, that should be resolved by breaking the cycle. Both cycle types result from conflicts, and should be broken. When conflicts do not occur frequently optimistic methods have an advantage. With different transactions loads (mixes of transaction types) one technique type (i.e., either optimistic or pessimistic) may provide better performance than the other.

Locking-based Concurrency Control

Strict two phase locking (S2PL) is a common mechanism utilized in database systems to enforce both conflict serializability and strictness (a special case of recoverability which allows effective database recovery from failure) of a schedule [25, 28, 192]. In this mechanism each data item is locked by a transaction before accessing it (any read or write operation): The item is marked by, associated with a lock of a certain type, depending on the operation (and the specific implementation). Various models with different lock types exist. Table 3.1 presents the different lock modes in a read/write transaction model. Since transactions can read or write data items, two types of locks, or *lock modes* are associated with every data element x: a *read lock rl(x)*, also known as *shared lock*, and a *write lock wl(x)*, also known as *exclusive lock*. It is common to describe the *compatibility* of locks in tabular form [192]. The table is meant to be read as follows: if a transaction T_i has set a lock $pl_i(x)$ and another transaction $T_j, j \neq i$, requests a lock $ql_i(x)$, then $ql_i(x)$ will be granted (i.e., it is compatible with $pl_i(x)$) if the corresponding table entry shows a $+$. Otherwise the lock will not be granted.

In some models, locks may change type during the transaction's life. As a result access by another transaction may be blocked, typically upon a conflict (the lock delays or completely prevents the conflict from being materialized and be reflected in the serialization graph by blocking the conflicting operation), depending on lock type and the other transaction's access operation type. Employing an S2PL mechanism means that all locks on data on behalf of a transaction are released only after the transaction has ended (either committed or aborted).

Mutual blocking between transactions may result in a deadlock, where the execution of transactions is stalled, and no transaction can proceed (and consequently commit). Thus deadlocks need to be resolved to complete these

		Lock requested	
		$rl_i(x)$	$wl_i(x)$
Lock held	$rl_j(x)$	+	-
	$wl_j(x)$	-	-

Figure 3.1: Lock Mode Compatibility

transactions' execution and release related locks. A deadlock is a reflection of a potential cycle in the precedence graph, that would occur without the blocking when conflicts are materialized. A deadlock is resolved by aborting a transaction involved with such potential cycle, and breaking the cycle. It is often detected using a wait-for graph (a graph of conflicts blocked by locks from being materialized; it can be also defined as the graph of non-materialized conflicts, since conflicts not materialized are not reflected in the serialization graph and do not affect serializability), which indicates which transaction is "waiting for" lock release by which transaction, and a cycle means a deadlock. Aborting one transaction per cycle is sufficient to break the cycle. Transactions aborted due to deadlock resolution are restarted and executed again immediately.

Non-locking Concurrency Control

Not all concurrency control algorithms use locks. Other techniques are **timestamp ordering**, **serialization graph testing**, and **commit ordering** [25, 28, 192]. Timestamp ordering assigns each transaction a timestamp and ensures that conflicting operations execute in timestamp order. The protocol works under the following assumptions:

- Every timestamp value is unique and accurately represents an instant in time;
- No two timestamps can be the same;
- A higher-valued timestamp occurs later in time than a lower-valued timestamp.

Serialization graph testing tracks conflicts and ensures that the serialization graph is acyclic. Commit ordering ensures that conflicting operations are consistent with the relative order in which their transactions commit.

3.1.2 Semantically Rich Operations and the Unified Theory of Concurrency Control and Recovery

The classical theory of transaction management contains two different aspects, namely concurrency control and recovery, which ensure serializability and atomicity of transaction executions, respectively. Although concurrency control and recovery are not independent of each other, the criteria for these two aspects were developed orthogonally and as a result, in most cases they are incompatible with each other. A unified theory of concurrency control and recovery for databases with read and write operations has been introduced in [12, 160] that allows reasoning about serializability and atomicity within the same framework. The unified theory of concurrency control and recovery has later been extended to provide a unified transaction model for databases with an arbitrary set of semantically rich operations in [181]. This theory concentrates mostly on the conflict behavior of semantically rich operations, which is more complex than those of the primitive read/write operations.

In distributed database environments a transaction is often considered as a partial order of different local sub-transactions. Each such sub-transaction can be in turn considered as an operation. These operations in general are not only read/write accesses on pages, but an arbitrary finite set of possible actions on data objects (data items, data elements), called *semantically rich operations*. In order to prove the correctness of execution of transactions consisting of semantically rich operations in a failure prone distributed database environment, the unified theory treats concurrency control and recovery uniformly, compared to the classical theory.

This approach is further based on the assumption that with each operation invocation an *undo* (or *inverse*) operation must be given. The purpose of the inverse operation is to remove from the database all the "recognizable" effects of the corresponding operation [181]. A sequence of operations α is said to be **well-formed** if each undo operation op^{-1} is preceded by its corresponding operation op.

Definition 3.12 ***(Effect-free Operations)*** *A sequence of operations σ is said to be* **effect-free** *if, for all possible sequences of operations α and β such that $\langle\ \alpha\ \sigma\ \beta\ \rangle$ and $\langle\ \alpha\ \beta\ \rangle$ are well-formed, the sequence of the return values of β in $\langle\ \alpha\ \sigma\ \beta\ \rangle$ is the same as in $\langle\ \alpha\ \beta\ \rangle$.* □

Following this definition op^{-1} is the undo operation for op if and only if the sequence $\mathrm{op}\ \mathrm{op}^{-1}$ is effect-free.

To unify concurrency control and recovery the authors defined commutativity for the undo operations. The unified theory is based on the (semantic) serializability with respect to the commutativity relation for regular and undo operations.

The classical read/write model of a database system defines a conflict between two operations if at least one operation is a write [28]. A more general notion of a conflict, applicable to the context of transactions consisting of semantically rich operations, is to consider two operation invocations as compatible if their execution order is irrelevant from an application point of view. The semantic property of the operations that we want to exploit is the compatibility of pairs of operations.

Definition 3.13 ***(Commutative Operations)*** *Let* op_1 *and* op_2 *be two operation invocations. Then,* op_1 *and* op_2 *commute if for any pair of sequences* α, β *of operation invocations the return values are the same in* $\langle\ \alpha\ \mathrm{op}_1\ \mathrm{op}_2\ \beta\ \rangle$ *and* $\langle\ \alpha\ \mathrm{op}_2\ \mathrm{op}_1\ \beta\ \rangle$. *Otherwise, the operation invocations are in conflict, i.e., (*op_1, op_2*)* $\in$ *CON with CON* $\subseteq \{O_{T_1}, ..., O_{T_n}\} \times \{O_{T_1}, ...O_{T_n}\}$ *being the conflict relation.*
$\square$

In transaction models based on the unified theory of concurrency control and recovery, the schedulers need to know about the *commutativity* of their operations. The notion of conflict is then defined based on the non-commutativity of the operations.

A correctness criterion called (prefix-) expanded serializability is introduced which allows efficient correctness testing based on serialization graph methods. At the same time, a class of schedules (called prefix reducible), which guarantees both serializability and atomicity in a failure prone environment was introduced. Several protocols were developed to generate such schedules by a database concurrency control mechanism. By representing all recovery-related actions explicitly in an execution, they can be treated as regular actions. The effect of recovery-related actions is then visible to the scheduler. This approach allows more correct executions.

3.1.3 Distributed Transactions

Although the transaction models introduced so far has proven very useful in traditional database applications (relatively short execution time, small number of concurrent transactions), they do not solve the problem of having to deal with transactions that span over long periods of time or multiple databases. The problem of distributed transactions (running transactions on multiple sites and updating resources within multiple resource managers) has been solved by allowing participants in a transaction to locally manage their resources, and in addition one of them should coordinate the actions of all the other transaction participants. This coordination has been achieved by using a *two-phase commit* protocol [95].

The Two-Phase Commit Protocol

For the distributed transactions to be ACID-compliant, all the transaction participants must enforce ACID properties. Traditionally, this is accomplished by using the two-phase commit (2PC) protocol, which centralizes the decision to commit.

During the initial (*prepare*) phase of the protocol, a transaction coordinator sends out a "prepare-to-commit"' message to all the transaction participants that have enlisted in the transaction (subordinates), requesting that each one of them indicate its readiness to commit or roll back the work managed in the scope of the given transaction. The subordinates may have spawned pieces of the transaction on other nodes, and in this case the "prepare-to-commit" message must be propagated (the transaction is now a tree with the coordinator at the root). For their part, the subordinates attempt to checkpoint their work and obtain locks for the affected resources and, if successful, respond with a "ready-to-commit" message. Otherwise, they issue a vote to roll back the transaction (a "refuse" message). The coordinator proceeds with the second (*commit*) phase only if all transaction participants have voted to commit, and after logging the information in a safe place. During this phase, the coordinator issues the appropriate command (commit or roll back) to all the subordinates. The transaction has completed when all the subordinates have safely committed their part and made it durable. It is important to note that the two-phase commit protocol is a blocking protocol. Once a subordinate receives the "prepare-to-commit" message and replies with a commit vote, it is obligated to lock the relevant records or data until the coordinator communicates an outcome during the second phase.

The X/OPEN Distributed Transaction Processing (DTP) has been proposed by the Object Management Group and a widely used open standard protocol for two-phase commit [197]. The X/OPEN XA specification describes the interface between a global transaction manager and local resource managers. The two major interfaces specified in the DTP model are the TX and XA interface. The XA Specification describes what a resource manager must do to support transactional access. At the same time, the XA specification allows participants to withdraw in the first phase of the two-phase commit if they do not have to update any resources.

The Paxos Commit Protocol

The Paxos commit [116] algorithm is based on an older Paxos consensus algorithm which is a protocol that allows a distributed system to reach a consensus, i.e., to agree on a value, such as a leader, despite the failure of some of them. Instead of a single transaction manager, the Paxos commit algorithm uses 2N+1 transaction managers. If any N of these transaction managers fail, the remaining N+1 can agree whether to commit or abort. The details are complex and are omitted here. However, the overhead in terms of the total delay and the total number of messages is small. Moreover, when N=0, the Paxos commit algorithm reduces to the classical two-phase commit algorithm.

Paxos can tolerate lost messages, delayed messages, repeated messages, and messages delivered out of order. It will reach consensus if there is a single leader for long enough that the leader can talk to a majority of participants twice. Paxos is an asynchronous algorithm; there are no explicit timeouts. However, it only reaches consensus when the system is behaving in a synchronous way, i.e. messages are delivered in a bounded period of time.

Lamport and Gray applied Paxos to the distributed transaction commit problem [94]. Paxos was used to effectively replicate the transaction managers of 2PC, and used an instance of Paxos for each participant involved in the transaction to agree whether that participant could commit the transaction. Paxos Commit does not use Paxos to solve the transaction commit problem directly, i.e. it is not used to solve uniform consensus, rather it is used to make the system fault tolerant.

3.1.4 Chained Transactions and Sagas

Despite their wide use, the flat transactions could not deal with a series of problems related to partial roll back of compound business transactions (only total roll back is possible), long running transactions, and/or transactions that spanned across companies or the Internet. As a possible solution to the above mentioned problems, mechanisms that extend the control flow beyond the linearity level of the flat transactions was proposed. One popular model was based on the idea of "chaining" sequences of atomic transactions - chained transactions or sagas [24, 84].

The simplest form of chained transactions use the concept of *syncpoints* (or savepoints) to allow periodic saves of the accumulated work. This mechanism allows to roll back the work while still maintaining a live transaction (which makes it different from a commit), but on the other hand a syncpoint is volatile (in the case of a system crash all the data are lost). A chained transaction can be defined as a back-to-back transactions (small, sequentially-executing sub-transactions), in which transaction context and locks can be passed on from one transaction to the next. A committed transaction triggers the next commit, until the whole chained transaction has committed. In case of a failure, the previously committed sub-transactions will have already made durable their changes, so roll back is only possible until the beginning of the most recently-executed sub-transaction.

Based on the idea of chained transactions, sagas were proposed in combination with a compensation mechanism to roll back. The saga transaction model [84] permits a long-lived transaction to be divided into a sequence of sub-transactions, each of which has an associated compensating sub-transaction that can be triggered to semantically undo the effects of its committed associate. If a saga sub-transaction fails and cannot recover, its partial effects are undone (backward error recovery), and a chain reaction occurs in which any successor committed sub-transaction of the same saga is (in reverse execution order) subjected to its respective compensation actions. These compensations do not necessarily return the system state to the point which existed when the saga began. Unlike the non-atomic chained transactions, that cannot undo the committed sub-transactions in case of an abort, the sagas use the compensating sub-transactions to return the system to a state that is equivalent to the start state from the application's point of view.

3.1.5 Layered Transactions

The major restriction of flat transactions is that there exists no possibility to take into account more sophisticated operations which in turn are defined by means of a sequence of more basic (read and write) operation, and consequently either commit or abort parts of a transaction. For this reason, the flat transaction model has been extended to introduce more dimensions of control. Nested transactions [126, 127] are a generalization of savepoints. Whereas savepoints allow organizing a transaction into a sequence of actions that can be rolled back individually, nested transaction form a hierarchy of pieces of work [95]. Nested transactions provide the ability to define transactions within transactions, the initial transaction is decomposed into subtransactions or child transactions, depending on the functionality they provide. This decomposition introduces a very important feature, which allows parts of a transaction to fail without the necessity to abort the entire transaction. The failure of a sub-transaction can be trapped by the parent transaction and retried using an alternative, still allowing the main transaction to commit. A child transaction can only start after its parent starts and a parent can only commit after all its children have committed. Each child is atomic, which means it can abort independently, determining an action from the parent. The parent can, in this case, trigger another sub-transaction to execute as an alternative. This mechanism allows nested transactions to maintain consistency.

Based on the mechanism of nested transactions there appeared later several other models, such as *multilevel transactions* (also called layered transactions) [20, 21, 189, 190], and their generalization *open nested transactions* [89, 191]. Multilevel transactions are a variant of nested transactions where all transaction trees have their levels corresponding to the layers of the underlying system architecture. This model introduces the pre-commit, allowing sub-transactions to commit before the root transaction actually commit. This addition to the model makes it impossible to roll back in a traditional way; instead compensating sub-transactions are used to semantically undo the work done before.

Based on the visibility of the sub-transactions within a nested transaction, the following classification can be made:

Closed Nesting: In closed nesting, a transaction may execute child transactions (sub-transactions). A parent does not execute while any of its children do, but in general a transaction may have multiple concurrent children [126, 127]. The effects of a committed sub-transaction are, in this model, only visible to its parent top-level transaction. Siblings or other concurrent transactions cannot see the changes made by sub-transactions until their parent top-level transaction has committed.

Open Nesting: Open nesting is, not surprisingly, similar to closed nesting. However, in the open nesting case the parent and child execute at different levels of abstraction. The restriction of the sub-transactions to be visible only within the scope of its top-level transaction is dropped in the open nested transaction model. This relaxation of the isolation property of transactions increases concurrency however, recovery becomes increasingly complex, as (semantical) compensating transactions are required in order to undo the effects of previously committed sub-transactions [89, 113].

The classical read/write model of a database system defines a conflict between two operations if at least one operation is a write [28]. A more general notion of a conflict is to consider two operation invocations as compatible if their execution order is irrelevant from an application point of view. This general notion of conflict has been introduced together with the more general concept of semantically rich operations.

In multilevel transaction models, the schedulers at each level need to know about the *commutativity* of their operations at the respective level. In this model, scheduling at each level is addressed separately. Therefore, the correctness criterion for a layered schedule is based on the one for flat transactions, with the extension that orders are preserved at the next level.

3.1.6 Multiversion Concurrency Control

We have so far worked under the assumption that there exists only one version of each database object. Consequently, each write operation on a data object x would overwrite its value, and each read operation on x would return the latest value of x. In such systems, transactions appear to be performed one at a time in some order. This is achieved by ensuring a serially equivalent interleaving of transaction operations. In a distributed system model, the main correctness criterion for replicated databases is *one-copy serializability (1SR)* [28]. The effect is that transactions performed on the database replicas have an ordering which is equivalent to an ordering obtained when the transactions are performed sequentially in a single centralized database.

Nevertheless, in practical applications and commercial database systems multiple versions of data exist at the same time. Multiple versions of data are used in database systems to support transaction and system recovery. These multiple versions of data can also be exploited to improve the degree of concurrency that is achievable in the system. The higher degree of concurrency can be achieved since tardy read requests can be serviced by reading appropriate, older versions of data. Thus, *read-only* transactions, which do not have any write operations are executed almost unhindered in most multiversion concurrency control schemes. At the same time, the adverse effect of concurrent *update* transactions on read-only transactions are minimized. The existence of multiple versions is visible only to the scheduler implementing the protocol, and not to the user transactions which refer to the object as x. The higher degree of concurrency is achieved due to the fact that there are no write/write or read/write conflicts.

In a multiversion database, each write operation on an object x produces a new *version* of x. Thus, for each object x in the database, there is an associated list of versions. A read operation on x is performed by returning the value of x from an appropriate version in the list. More formally, the operations of transactions refer to database objects (data items), and not versions. It is the responsibility of the underlying database system to map each read operations $r_k(x)$ to a version x_i, such that the write operation $w_i(x_i)$ is executed before $r_k(x)$, in other words, to ensure that serializability is guaranteed. This is referred to as *version order* or *version function* and has to be transparent to the user. Version ordering offers a higher degree of freedom.

Definition 3.14 ***(Multiversion Schedule)*** *A multiversion schedule is a schedule in which every operation is associated with a corresponding version of a database object.* □

As previously mentioned, versions should be transparent to the applications. Thus, it is important that as far as the users can tell the behavior of a multiversion schedule is the same as a serial schedule over the same set of transactions executed over a single version database. In this case we call the multiversion schedule to be one-copy serializable (since one-copy serializability is the commonly accepted correctness criterion for multiversion databases [28]). In order to prove the correctness of a multiversion schedule a multiversion serializability graph has to be constructed.

Definition 3.15 ***(Multiversion Serialization Graph)*** *Let S be a multiversion schedule over τ and $\lhd$ a version order. The multiversion serialization graph of S, $MVSG(S, \lhd)$ is a graph for which each transaction $T \in \tau$ is an node in the graph and the following relations hold:*

1. *For each $w_i(x_i)$ and $r_k(x_i)$ in S with $w_i(x_i) \ll_s r_k(x_i)$, there exists $T_i \rightarrow T_k$ an edge in $MVSG(S, \lhd)$ (conceptually equivalent to a write/read conflict in a normal serialization graph).*

2. *For each $w_i(x_i)$ and $r_k(x_i)$ in S, $i \neq m \neq k$ and $x_i \lhd x_m$, there exists $T_i \rightarrow T_m$ an edge in $MVSG(S, \lhd)$, otherwise (if $x_m \lhd x_i$) there exists $T_k \rightarrow T_i$ an edge in $MVSG(S, \lhd)$.*

□

Based on this we can define the following correctness criterion for multiversion databases:

Theorem 3.16 ***(Multiversion One-Copy Serializability)*** *A multiversion schedule M is serializable if and only if $MSVG(M, \lhd)$ is acyclic.* □

A proof of this theorem can be found in [28].

Several concurrency control algorithms have been developed for multiversion databases [7, 25, 47, 48, 154, 188]. Multiversion timestamp ordering was introduced in [154]. This approach has the advantage that read requests are never rejected, and hence transactions consisting entirely of read operations are never aborted. Multiversion two-phase locking was originally proposed in [47]. This protocol makes a distinction between read-only and update transactions before transactions begin execution. Update transactions are executing according to the standard two-phase locking protocol, while read-only transaction are executed immediately without delays. [188] proposed several protocols to implement read-only transactions and to manage multiversion databases. In particular, several techniques to discard old versions of data are described. In [7], a technique is proposed to decouple the version control mechanism from the concurrency control protocol. Read-only transactions do not have any concurrency control related overhead in this scheme.

Snapshot Isolation

The snapshot isolation protocol is an example of a protocol that considers several versions per object - as each transaction works with its own snapshot. The snapshot isolation protocol [23, 76] guarantees that all reads made in a transaction will see a consistent snapshot of the database (in practice it reads the last committed values that existed at the time it started), and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot. A transaction executing under snapshot isolation appears to operate on a personal snapshot of the database, taken at the start of the transaction. When the transaction concludes, it will successfully commit only if the values updated by the transaction have not been changed externally since the snapshot was taken. Such a write-write conflict will cause the transaction to abort.

In contrast to serializable protocols, snapshot isolation permits write skew anomalies. In a write skew anomaly, two transactions (T_1 and T_2) concurrently read an overlapping data set, concurrently make disjoint updates and finally concurrently commit, neither having seen the update performed by the other. Were the system serializable, such an anomaly would be impossible, as either T_1 or T_2 would have to occur "first", and be visible to the other. In contrast, snapshot isolation permits write skew anomalies.

3.2 Replication Management

Replication is one of the key requirements of almost every enterprise application to meet its goals of availability, scalability and performance. Managing the data replication in a large heterogeneous environment that supports multiple applications, such as data Grids, is a big challenge. Typically applications deal with different types of data and have different requirements for data availability, performance and consistency; hence they need different replication technologies to replicate their data.

Although the replication of persistent data is crucial to load sharing and achieving high availability and scalability, it introduces the complexity of consistency management. Replication is not a simple redundancy or backup mechanism. Rather, it involves a fully transparent full or partial distribution of copies of data objects (also referred to as *replicas*), for sites that are dispersed over possibly very wide areas. In particular, it includes a wide range of protocols for replication and failure recovery as well as policies to trade off requirements of availability and consistency of replica in transaction-intensive systems such as OLTP databases. Similar to multiversion concurrency control, the corresponding correctness criterion is *one-copy serializability* [26, 27].

By analogy to the expectations from any distributed information systems [45], replication management is expected to fulfill, as closely as possible, the following requirements:

Scalability: Replication management scheme should be able to handle a large number of replicas and simultaneous replica creation while at the same time scaling in size with regard to the number of copies.

Performance: Replica management should be performed in a timely manner and with a reasonable amount of resources. At the same time the response times should not be affected by replica decisions. In general many systems can achieve a good query performance. In contrast updates generally do not become faster with replication.

Consistency and Correctness: The main goal of replication management is to guarantee data consistency and the correct concurrent execution of transaction. In practice, an environment where updates to a replica are needed, different degrees of consistency and update frequencies should be provided. Many existing approaches sacrifice correctness in favor of better performance.

Reliability and Availability: Replication management should include failure handling and recovery techniques such that the failure of a node does not affect the overall correctness or performance.

3.2.1 Classification of Replication Mechanisms

In databases, replica control mechanisms ensure data consistency between the copies. Gray et al. [92] categorize these mechanisms according to two parameters: the first one refers to which copies can be updated and the second one refers to when updates are propagated. Figure 3.2 exemplifies these strategies by contrasting propagation strategies with ownership strategies.

Propagation/ Ownership	**Eager**	**Lazy**
Primary Copy	1 transaction 1 object owner	N transactions 1 object owner
Update Everywhere	1 transaction N object owners	N transactions N object owners

Figure 3.2: Replication Taxonomy

Update Location

The first parameter is *update location*, also called *ownership* [92], and defines the permissions for updating copies. In regard to update location, a *primary* approach only allows data to be updated in one primary site. This parameter decides where the updates can take place and imposes no restriction on queries. Thus, in a primary approach, if a client submits updates to a site other than the primary site, the updates will be either refused or redirected to the primary site for execution. Different data items might have different primary sites. In this case, however, transactions that want to update data items with different primary sites are disallowed. Read-only transactions are allowed at any site. In contrast, in an *update everywhere* approach the updates are accepted and executed at the local site to which the updates are submitted. In general, update everywhere approaches are more flexible than primary approaches. Since the primary copy approach requires all updates to be performed first at one copy (the primary or master copy) and then at the other copies, this simplifies replica control at the price of introducing a single point of failure and a potential bottleneck. The update everywhere approach allows any copy to be updated, therefore speeding up access at the price of making coordination more complex.

Update Propagation

Irrespective of the update location, updates must be propagated to other sites. The second parameter is *update propagation* and defines when the updates are propagated [92]. The update (synchronization) of replicas can be done in two ways. The first approach comes from standard database technology and it is known as *eager (synchronous) replication*. Using an eager approach update propagation must happen before the transaction commits, thus within the transaction boundary. An eager approach provides strong consistency because a transaction will not commit until it is certain that it will be able to commit in all other available sites. However, this delays transaction execution. The other approach is known as *lazy (asynchronous) replication*. By contrast, a lazy approach allows update transactions to commit before propagating the updates to other sites. The updates are later propagated to the remaining replicas by decoupled refresh transactions. A lazy approach provides therefore only weak consistency because of early commit, but transaction response times in a lazy approach are lower than in an eager approach. Compared to the eager approach, the lazy replication approaches require additional efforts to guarantee serializability.

3.2.2 Eager Replication

Early research in replication addressed eager approaches since they provide strong consistency and a high degree of fault-tolerance. As mentioned before eager replication is the simplest way to achieve one-copy serializability: despite the existence of multiple copies, the effect of transactions performed by clients on replicated objects should be the same as if they had been performed on a single set of objects (also called *one-copy equivalence*) and the execution of concurrent transactions is coordinated so that it is equivalent to a serial execution over the logical copy (serializability). Eager replication typically employs distributed transactions, which use an atomic commitment protocol (such as *two-phase commit*). All the changes are made visible synchronously when the original transaction commits. Early solutions used such eager primary copy approaches [13, 172]. Later algorithms followed an update everywhere approach based on quorums, e.g. read-one-write-all (ROWA) [28] or read-one-write-all-available (ROWAA). Efforts have been devoted to optimizing quorum sizes [5, 52]. More recently, epidemic protocols have been proposed [6] in which communication mechanisms providing causality are augmented to ensure serializability.

Postgres-R(SI) [196] presents a prototype that integrates replica management with database concurrency control. This solution does not require declaring transaction properties in advance as previous attempts in the same direction, but its replication algorithm must be implemented inside the DBMS. Thus, this approach imposes significant obstacle to the portability and deployment in a Grid environment, the interoperability in networks of systems based on products of different vendors, and is difficult to maintain even for releases of new versions from the same brand. SI-Rep [119] provides a solution similar to Postgres-R(SI) on top of PostgreSQL which needs the write set of a transaction before its commit. Write sets can be obtained by either extending the DBMS, thus compromising DBMS autonomy, or using triggers.

This requires declaring additional triggers on every database table, as well as changing triggers every time the database schema is altered.

Eager Primary Copy Replication

In an eager primary approach update transactions are only allowed to execute at the primary site which performs traditional concurrency control to isolate conflicting transactions. As long as other sites apply and commit updates in the same order as at the primary site, data will be consistent. Furthermore in this approach the changes of transactions are propagated eagerly (i.e., before commit). The originating transaction is only then committed when the update has been performed on all replicas, typically by means of a two-phase commit protocol [74, 95]. Due to the fact that concurrency control takes place at the primary copy, query transactions always see the latest version of data objects. Systems such as distributed INGRES [172], employ such approaches.

Eager Update Everywhere Replication

Update everywhere approaches do not require update transactions to be submitted or forwarded to a primary site for execution, but still propagate the updates within the same transaction. However, it is more difficult to keep data consistent than in primary approaches. This is due to the fact that in a primary approach conflicts between update transactions are detected in a single site (i.e., the primary) while in an update everywhere approach conflicting update transactions can run concurrently on different sites. When using distributed locking, an item can only be updated after it has been locked on all sites. Thus, an update everywhere approach requires additional coordination between different sites for concurrency control purposes, which is not trivial. Gray et al. [92] claim that update everywhere approaches may lead to high deadlock (directly proportional to n^3, n being the number or replicas) and high abort rates if many transactions run concurrently on different sites. The challenge of eager update everywhere approaches is to provide replica control in order to guarantee global transaction isolation.

Read-One-Write-All-Available

One solution to that is used to deal with site failures is *Read-One-Write-All-Available* [26, 28] and it is the most known of the quorum free protocols. The protocol requires all *available* replicas to participate in the write quorum. Read-only transaction can be served by any replica. When a failed site reconnects again, it is synchronized with one of the most up-to-date sites. Nevertheless, ROWAA approaches are not able to deal with network partitioning.

Majority Consensus Voting

Other approaches to improve availability are based on quorum protocols. Quorum protocols offer several benefits when used to maintain replicated data [141, 175], for example, updates are required only for a subset of replicas. In these approaches, write and read quorums are subsets of replicas used to perform write and read operations, accordingly. By guaranteeing that read and write quorums intersect, any read operation will include the most recent version. Voting type algorithms are able to guarantee the consistency of replicated data in the presence of both site failures and network partitions, as opposed to other types of algorithms that can only handle site failures. Quorum-based protocols have also been many times proposed as an alternative for data replication. But most quorum-based protocols impose a logical structure on the nodes such as [5], which can soon become unfeasible in infrastructures like the Grid. Moreover, [108] concluded that a ROWA approach is better for a large range of database applications than a quorum-based approach as it shows better scalability. The result is significant as quorums are often suggested to reduce the overhead of scale replication. ROWA is a basic solution to enforce strong replica consistency by means of eager replication [133]. This approach (conceptually similar to our replication protocol for the update sites) requires that whenever a transaction updates a replica, it also updates all other replicas (using distributed transactions), thereby enforcing the mutual consistency of the replicas. Contrary to our approach however, the atomic commitment of the distributed transaction in the ROWA approach typically relies on the two-phase commit protocol [133] which is known to be blocking (i.e., it does not deal well with nodes failures) and has poor scale up.

Atomic Broadcast Replication

Atomic broadcast replication is an interesting approach to ensure fault-tolerance in distributed replication systems and combine eager replication with group communication primitives [110, 111, 161]. The approach consists in providing transactions with a communication primitive that allows them to broadcast and deliver messages in such a way that transactions agree not only on the set of messages they deliver but also on the order of message deliveries. A group communication ensures therefore a total order on all delivered messages, such that the global transaction manager can order conflicting operations. When a transaction is submitted at a site, the request is broadcasted to all replicas by using atomic broadcast. A blocking protocol that would guarantee consistency is not needed, because of the total order provided by the atomic broadcast. Such approaches need to be directly supported by the database management system which would not be feasible when using off-the-shelf database components. Clustered database replication requires reliable multicast with total order to ensure that each replica applies updates in the same order. Even though various optimizations have been developed, the group communication layer is an intrinsic scalability limit for such systems. Recent efforts have tried to extend multi-master replication to WAN environments [120]. However, these approaches are still suffering from network latency and unreliability of long distance links are still making it impractical to have any reasonable production implementation of fast reliable multicast.

3.2.3 Lazy Replication

As already seen eager replication makes it relatively easy to guarantee transactional properties, such as serializability. However, when the transactions are distributed and relatively long-lived, the approach does not scale well. Lazy (or asynchronous) schemes, on the other hand, update replicas using separate transactions. Lazy replication techniques asynchronously propagate replica updates to remaining sites in the system after the original transaction commits. Using this approach, the replicas continue to provide service regardless of site failures and network partitions. The mechanism used by lazy replication approaches works as follows: each time a transaction updates a replica at some site, all the updates are propagated towards the other replicas of the same object, and finally these replicas are updated in separate refresh transaction. Consequently, this scheme looses the mutual consistency property as ensured by 2PC. The interval of time between the execution of the original update transaction and the corresponding refresh transactions may be large due to the time needed to propagate and execute the refresh transactions. There are two strategies that can be used to schedule refresh transactions. The first strategy begins the propagation of the refresh transactions *immediately* after the first update of the original transaction. In this approach updates are propagated as soon as they occur, but always as independent transactions. The major drawback of these approach is that updates are propagated before the end of the original transaction and aborts can be expensive. The second strategy *defers* the propagation of the updates until the commit of the original transaction. The major advantage of this approach is that changes of multiple transactions can be applied together as bulk refresh transactions [10, 157]. In this case recovery also becomes simpler since it only involves local copies. As mentioned before, in order to reduce latency, lazy replication updates all the copies in separate transactions after the commitment of the initial transaction. However, lazy replication implies that during a certain time, copies of the same data diverge: some have already the new value introduced by the initial transaction, others have not. In this case the concept of *freshness* is used to measure the deviation between replica copies. This divergence refers to the notion of data freshness: the lower the divergence of a copy with respect to the other copies already updated, the fresher is the data copy. Users may accept to read stale data, i.e. data not perfectly fresh.

Freshness issues have been addressed in several approaches [10, 157, 138, 137], where the concept of freshness is typically used to measure the deviation between replica copies. The proposed protocols allow the users to specify freshness requirements as a quality of service parameter attached to their queries which defines what staleness of data is acceptable for the user. Most importantly, transactions always access consistent data irrespective of their freshness requirements. However, [157] relies on full replication of the databases and does not allow distributed execution of queries. The protocol proposed in [10] overcomes these drawbacks but relies on a central component to serialize the updates.

Many replication solutions for databases already exist in the literature that may be partially applicable to grid environments, such as [10, 14, 135]. In [10] a protocol for database replication that supports freshness and lazy update propagation for many read-only nodes is provided. In [9], the protocol from [10] is adapted for data Grid environments. However, this protocol makes a strong assumption about the existence of a central component which is used to collect and serialize all updates at the update nodes. In fact, this is conceptually equivalent to having only one update node in the system, which is a potential bottleneck and a single point of failure, and therefore not practical in a Grid environment. In [14], several updateable replicas are supported but a single, global replication graph is required which, in turn, requires a single site where the graph is located. Consequently such a site becomes a single point of failure.

In [119] a centralized middleware is developed that forwards each transaction to one of the replicas for execution. In [135] the protocol works under the assumption that one replica copy is designated as the primary copy, stored at a master node, and that update transactions are only allowed on that replica. Thus, none of the existing replication protocols can be fully decentralized which is an important requirement in infrastructures like the Grid.

Lazy Primary Copy Replication

In lazy primary copy protocols, the accesses are always managed by the same replica, such a replica may use any local concurrency control approach for avoiding conflicts between transactions. Primary copy solutions have been used in some commercial database systems. In a lazy master replicated database, a transaction can commit after updating one replica copy (primary copy) at some master node. After the transaction commits, the updates are propagated towards the other replicas (secondary copies), which are updated in separate refresh transactions. A central problem is the design of algorithms that maintain replicas consistency while at the same time minimizing the performance degradation due to the synchronization of refresh transactions. Finally, lazy primary approaches pose serious problems in case of failures. In case that the primary site crashes before propagating a change but after the commit of a transaction, the transaction will not be applied in the secondary sites. As [55] have shown in lazy primary copy schemes, serializability cannot be guaranteed without restricting the placement of primary and secondary copies in the system. Older work [32, 134] has attempted to minimize this limitation. A major drawback of these approaches is that transactions cannot update data items whose primary copies reside on different sites and in real applications especially in clusters, the complexities and limitations on replica placement are likely to be a significant liability.

Lazy Update Everywhere Replication

Lazy update everywhere replication permits clients to access any arbitrary site and to update any local data. These updates are later on propagated to the other replicas as independent refresh transactions. Since two or more sites might be applying conflicting transactions at the same time, this approach makes it very difficult to guarantee correctness. Using this approach replicas at different sites might not only be stale but also inconsistent. In such cases, lazy update everywhere replication requires *reconciliation* to decide which transactions are undone. Strategies for conflict detection and reconciliation include pre-arranged patterns like latest update wins, node priority or largest value. In some cases, however, manual reconciliation may be required. For this reason, some approaches have dropped the one-copy serializability criterion and have concentrated on relaxed data coherency mechanisms.

3.2.4 Replication with Relaxed Data Consistency

In some cases the one-copy serializability criterion may prove to be impractical, for example for certain high performance applications. Relaxing the mutual consistency of data leads to performance enhancement. Using this as a starting point, two main classes of approaches to relaxed data coherency have been developed. The first class is called *implicit* relaxed data coherency, as the coherency relaxation is not expressed in quantitative terms. Example of such approaches are: *epsilon serializability* [152, 168] and *epidemic replication* [6, 66, 149]. The second class of relaxed data coherency approaches is called *explicit*. In these approaches the specified coherency threshold has to be met. Examples of such approaches are: *identity connections* [159, 166] and *quasi copies* [83, 164].

3.3 Transaction and Replication Management in Grid Infrastructures

This section presents extended transaction models, such as web service transactions or grid transactions, as well as a discussion of Grid replication protocols.

3.3.1 Web Service Transaction Models

Although most classical transaction systems are based on implementations of the ACID transactions, the various properties of an ACID transaction can be relaxed to provide what are typically referred to as *extended transactions*; for example, an extended transaction model may relax atomicity to allow partial sets of participants to commit or abort, or it may relax isolation to allow concurrent users to observe partial results. Composing certain activities from long-running ACID transactions can reduce the amount of concurrency within an application or (in the event of failures) require work to be performed again. For example, there are certain classes of application where it is known that resources acquired within a transaction can be released early, rather than having to wait until the transaction terminates; in the event of the transaction canceling, however, certain activities may be necessary to restore the system to a consistent state (perhaps performing compensation or counter-effects). Such compensation and fault-handling activities (which may perform forward or backward recovery) will typically be application-specific, may not be necessary at all, or may be more efficiently dealt with by the application. Thus an extended transaction model is more appropriate for long-duration interactions.

The division of a transaction into activities (atomic units of work), defining alternative paths and compensation activities are among the techniques used to circumvent some of the problems posed by distributed systems (blocking resources for a long period of time or a lot of finished work that needs to be rolled back in case of failure).

All these models offer a higher level (application level) transaction support.

The Business Transaction Protocol

The latest version of the Business Transaction Protocol [38] has been released by the OASIS consortium in 2009.

BTP defines a protocol that makes minimal assumptions about the implementation structure and the properties of the protocols that define how the transmission of BTP messages occurs. It defines protocol exchanges to ensure the overall application achieves a consistent result. This consistency may be defined a priori: all the work is confirmed or none at all (an atomic business transaction or atom); or it can be determined by application intervention in the selection of the work to be confirmed (a cohesive business transaction or cohesion).

The BTP protocol is considered to be an "open-top" protocol, with two distinct phases, in contrast to the traditional 2PC protocol, which is considered "closed-top" because it only admits a couple of commands. The first phase of the transaction protocol is called "provisional-effect" (requiring provisional or tentative state changes). The second phase of the protocol defined the termination of the transaction either by confirmation ("final effect") or by cancellation ("counter-effect"). How these phase are implemented is dependent on the implementation of the applications.

The commands used by BTP offer richer semantics and abstract from the background implementation. Another very important difference with respect to the traditional 2PC protocol is that the time between the two phases is entirely under the application's control.

WS-Coordination and Transaction

In reaction to the BTP protocol, Microsoft, IBM and BEA released a new set of specifications, called WS Coordination and Transaction [195] aimed at the reliable and consistent execution of web based business transactions using different interconnected web services.

WS-Coordination and WS-Transaction form together an extensible framework for providing protocols that coordinate the actions of distributed applications, by means of a coordinator. The WS-Coordination framework enables participants to reach consistent agreement on the outcome of distributed activities. The coordination protocols that can be defined in this framework can accommodate a wide variety of activities, including protocols for simple short-lived operations and protocols for complex long-lived business activities. It is worth mentioning that the use of the coordination framework is not restricted to transaction processing systems; a wide variety of protocols can be defined for distributed applications. However, so far WS-Transaction is the only specification of a protocol based on WS-Coordination.

The protocols specified by WS-Transaction are WS-AtomicTransaction and WS-BusinessActivity. The WS-AtomicTransaction specification is focussed on the existing transaction systems and protocols with strict ACID requirements and implements different flavors of the 2PC protocol. The WS-BusinessActivity specification provides flexible transaction properties and is designed specifically for long-duration interactions, where traditional transaction behavior (such as holding on to resources) is impossible or impractical.

WS-Composite Application Framework

Web Services Composite Application Framework (WS-CAF) was developed by SUN, Oracle, Arjuna, IONA and Fujitsu in 2003 [194]. It has been afterwards adopted as a standard by OASIS. It is a layered framework consisting of the following three parts: WS-Context (WS-CTX), WS-Coordination Framework (WS-CF) and WS-Transaction Management (WS-TXM).

All these specifications have been intended to be used alone, nevertheless, the coupling of these specifications can facilitate the construction of applications that combine multiple services together in composite applications.

The fundamental capability offered by the WS-Coordination Framework specification is the ability to register a web service as a participant in some kind of domain specific function. WS-Coordination Framework permits various coordination protocols to be layered on it. Web Service Transaction Management defines three protocols that are plugged into WS-CF and can be used with a coordinator to negotiate a set of actions for all participants that are to be executed based on the outcome of a series of related Web services executions. Examples of coordinated outcomes include the classic two-phase commit protocol, a three phase commit protocol, open nested transaction protocol, asynchronous messaging protocol, or business process automation protocol.

It is important to notice that despite of the flexibility that these models offer, all of them rely on a central coordination framework and/or atomic commitment protocols such as two-phase commit, which makes them not suitable for Grid environments.

3.3.2 Grid Transaction Models

Despite its importance, very little work has been done in the field of grid transactions. TM-RG (GGF Transaction Management Research Group) is working on Grid transactions with the goal of investigating how to apply transaction management techniques to Grid systems. This group is investigating possible Grid transaction approaches, based on relaxed isolation transaction models, such as the ones presented in the previous section.

The architecture GridTP, developed at the Shanghai Jiang Tong University [147], is based on the Open Grid Services Architecture (OGSA) platform and the X/Open DTP model, providing a consistent and effective way to make available autonomously managed databases in the Grid. This means that a two-phase commit protocol is used to atomically commit distributed Grid transactions.

A decentralized serialization graph testing protocol that ensures concurrency control and recovery in peer-to-peer environments has been proposed in [101, 177]. The proposed protocol ensures global correctness without relying on a centralized global serialization graph. Each transactional process is equipped with partial knowledge that allows the transactional processes to coordinate. Globally correct execution is achieved by communication among dependent transactional processes and the peers they have accessed. In case of failures, a combination of partial backward and forward recovery is applied. Although the paper is proposing a model based on known techniques, such as serialization graph testing and partial roll backs, it combines old techniques for a new purpose, presuming globally correct execution of concurrent transactions based on the local knowledge of the transactions. A similar approach has been recently adopted for peer-to-peer transaction management [17]. However, the proposed protocols do not handle replication and data versioning.

3.3.3 Grid Replication

Although considerable work has been done in the field of data management in the Grid, currently available Grid solutions use a simple user-initiated replication model. In general, a central catalog exists where all replicas of a file are registered. Single instances of centralized components could thus easily become a single point of failure in the system. When a file is uploaded and registered in the catalog, it is the responsibility of its owner to decide where and in how many copies the file should be replicated.

The European DataGrid (EDG) project was charged with providing a Grid infrastructure for the massive computational and data handling requirements of several large scientific experiments [70]. The design of the replica management system is modular, with several independent services interacting via the Replica Manager, a logical single point of entry to the system for users and other external services. The Replica Manager coordinates the interactions between all components of the systems and uses the underlying file transport services for replica creation and deletion. The replica management follows a simple user-initiated replication model, which puts the user in charge of replica creation and placement. Another decision taken by the user is the number of replicas to be created. Leaving this decision at the discretion of the user might lead to an uncontrolled increase and prove to be impractical. Since much of the coordination logic occurs within the client, asynchronous interaction is not possible and in the case of failures on the client side, there is no way to automatically re-try the operations. User feedback showed that the file catalog infrastructure was too slow both for inserts and for queries. Missing functionality identified included lack of support for bulk operations and transactions. It also became clear that queries were generally based on metadata attributes and were not simple lookups of a file's physical location.

To address these problems, the LHC Computing Grid has designed a new data management component, the LCG File Catalog (LFC) [71]. The LFC moves away from the Replica Location Service model used in previous LCG releases, towards a hierarchical filesystem model and provides additional functionality, such as cursors for large queries, allows timeouts and retries on the client side, as well as fixing performance and scalability issues seen in the EDG catalogs. Nevertheless, LFC is still deployed as a centralized component which is single point of failure in the system. Moreover, the middleware does not offer any integrated replication system. In general, replica management is done by the user using command-line tools.

Another replica management approach is the Globus Data Replication Service (DRS), whose function is to replicate a specified set of files onto a local storage system and register the new files in appropriate catalogs [87]. DRS builds on lower-level Grid data services, including the Globus Reliable File Transfer (RFT) service and Replica Location Service (RLS). As in previous approaches, replica management decisions are the responsibility of the user. At the same time, it is a general tendency in Grid replication mechanisms to ignore replica consistency and freshness of data. The same tendency can be observed in the DRS.

The Storage Resource Broker (SRB) [169] is a data Grid middleware software system that provides a uniform interface to heterogeneous data storage resources over a network. SRB supports both synchronous and asynchronous replication of files registered in the catalog. The system has much more functionality than existing replication system developed for the Grid. Nevertheless the synchronization of replicas is still triggered by the user and may suffer from consistency problems in case of multiple catalogs and while accessing more than one replica. Furthermore, it does not provide a dynamic replica management functionality. The dynamic selection of a replica is in general an issue that is not properly addressed in any available Grid replication tools.

3.4 Summary

The purpose of replication management approaches is to ensure fault-tolerance and availability while at the same time facilitating the users' access to data by improving query performance. At the same time, data consistency and correctness are important aspects which have to be guaranteed. In this chapter we have introduced the foundations of transaction and replication management in order to establish the basis for appropriate replication protocols.

After surveying a variety of replication protocols we are still missing an approach suitable for data Grid environments which is capable to seamlessly combine scalability, global correctness and quality of service guarantees. Previous research in the field of transaction and replication management have motivated our approach. In an environment where updates are infrequent, an optimistic concurrency control is combined with eager replication protocols to ensure consistency among update replicas. A lazy replication protocol is used to asynchronously propagate changes to read-only sites. Such an approach is thus a careful combination of eager and lazy replication protocols and can reproduce the performance characteristics of asynchronous replication management while still ensuring data coherency. The corresponding protocols are presented in the subsequent chapters.

4

The Re:GRIDiT Approach to Replication Management in a Data Grid

The overall goal of this thesis is to present a replication mechanism that combines scalability, global correctness and quality of service guarantees in a dynamic way. Chapter 2 presented detailed use case scenarios from various eScience domains which urgently require new integrated approaches to dynamic replication in a data Grid. Our Re:GRIDiT protocol dynamically manages replicas in the Grid, while at the same time providing freshness and correctness guarantees. The Re:GRIDiT family consists of three different protocols which target the three main problematic aspects identified in current data Grid infrastructures. These protocols are:

Re:SYNCiT synchronizes updates to several replicas in the Grid in a distributed way. The need for the Re:SYNCiT protocol is manifold: user application requirements have stressed the need for consistency: distributed concurrency control is required when user operations span several sites, i.e., support for distributed transaction management when data which are read or updated in a single transaction are distributed across several sites. At the same time replication management is needed since availability is highly important in such widely distributed environments as the data Grids. We employ a combination of both eager and lazy replication protocols that are capable of taking into account different levels of freshness. Our approach assumes no global coordinator: we enforce globally serializable schedules in a completely distributed way without relying on a central coordinator with complete global knowledge.

This is an important feature in order to allow for the application of data replication at Grid scale. Last but not least, we support a flexible data model in which we distinguish between mutable and immutable data objects. Mutable data objects can be updated. Immutable data objects, on the other hand, cannot be modified; once created they are kept until deleted, but several versions of the same immutable data object may exist. To the best of our knowledge this distinction between data objects has not been made in any available protocol, yet it is a straightforward consequence of the nature of many Grid applications.

Re:LOADiT is our approach to dynamic replica deployment and management. We build a system where user requests can be directed and executed by any replica, and where parameters such as load, freshness or network distance to the replica are used to determine a request's destination. We distribute data objects among several replicas to increase throughput and move frequently used/heavy accessed data objects to relatively inactive replicas, where they do not compete against each other for resources, and requests can be handled faster. Based on a combination of local load statistics, proximity and data access patterns, Re:LOADiT dynamically adds new replicas or removes existing ones without impacting global correctness.

Re:FRESHiT allows read-only clients to state how up-to-date their data can be. Users may demand a certain freshness level or a certain version and this includes the special case where users always want to work with up-to-date data. It supports the freshness-aware routing of queries in the Grid and also takes into account the sites' local load for replica selection without relying on any central component. In parallel, updates are propagated from the update sites to the read-only sites along the site hierarchy in a consistent manner.

In short, Re:GRIDiT was developed as a response to the need of a protocol that meets the challenges of replication management in a data Grid, and that re-defines the Grid and re-discovers it (in other words, that "re:grids it"), bringing it to a level where it can satisfy the needs of a large variety of users from different communities. Despite previous research in the field distributed transaction and replication management new approaches that seamlessly provide all these functionalities do not exist yet.

In this chapter we lay the basis for the Re:GRIDiT family of protocols and describe a system model for our data Grid replication system. The system addresses data replication in the Grid, in a completely distributed way, taking into account particular requirements coming from both the Grid infrastructure and the Grid applications, ensures correctness even in case of failures, and supports the Re:GRIDiT protocols. Consequently, in Chapters 5, 6 and 7 we present new protocols that allow us to correctly synchronize distributed updates to replicated data in the Grid (Chapter 5), dynamically manage replicated data in the Grid based on user access patterns (Chapter 6), and subsequently propagate changes to read-only replicas with freshness and correctness guarantees (Chapter 7). These protocols rely on functionality provided by the underlying system, as discussed in the following.

4.1 Architectural Layers

Despite recent advances, data Grid technologies often propose only very generic services for deploying large scale applications such as user authorization and authentication, data replication, fast and reliable data transfer, and transparent access to computing resources. In this context, we are facing important challenges such as the ones coming from the earth observation application presented in Section 2. We built therefore further functionality on top of the underlying Grid middleware, while taking into account the particular requirements coming from both the Grid infrastructure and the Grid applications (such as dynamically distributed management of replicated data, different levels of freshness, the absence of a central coordinator).

For this purpose, we envisage a layered architecture, consisting of three different *layers* (levels). Figure 4.1 sketches our architectural system. The bottom layer is built by computing nodes, storage nodes, fast network connections, in short any basic infrastructure which provides file management capabilities and a relational database system. At the middleware layer, Grid services provide homogeneous and transparent access to the underlying heterogeneous components while Re:GRIDiT services (which build on top of any Grid middleware) provide dynamic and distributed replication of Grid-enabled applications with data sharing and distributed computation capabilities in a way that is transparent to the applications and the users. The application specific layer provides high level and domain specific services taking into account the data semantics (such as collections of images), high level services (such as earth observation imagery, sophisticated image interpretation services), support for parallel and interactive applications, etc. The components in each layer build on new capabilities and behaviors provided by the lower layer. This model demonstrates flexibility and shows how Grid architectures can be extended and evolved upon. By utilizing Grid technologies to create a high-performing and scalable platform, we can easily build flexible and dy-

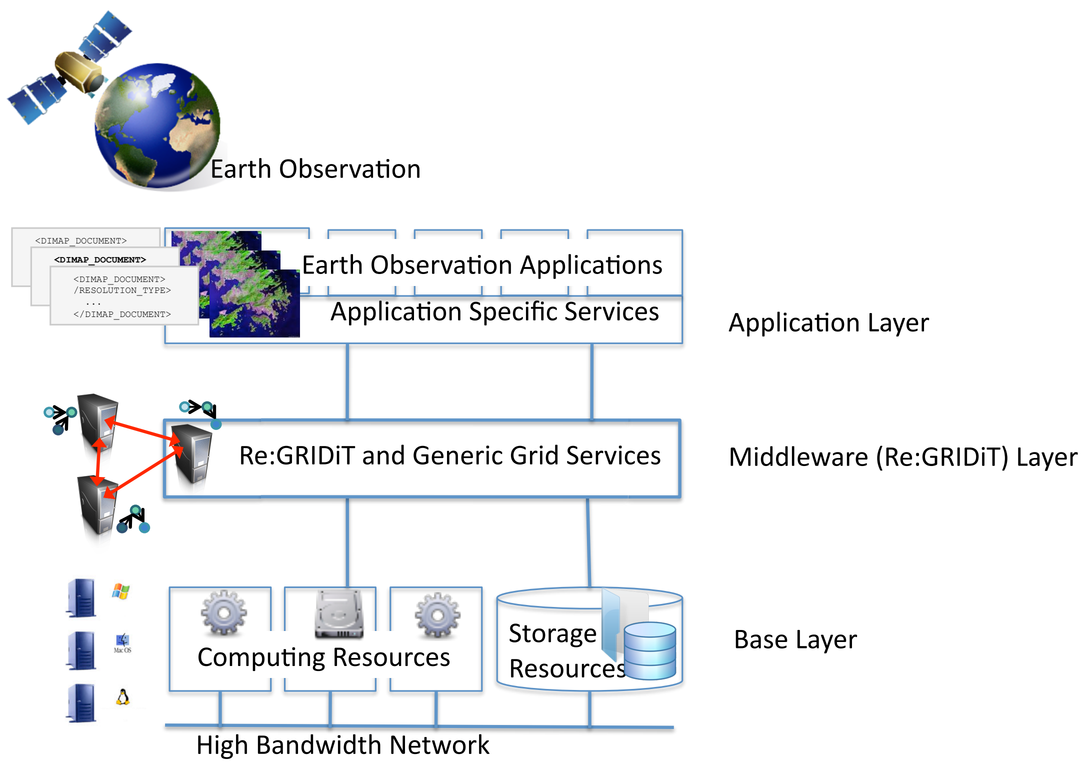

Earth Observation
Earth Observation Applications
Application Specific Services
Application Layer
Re:GRIDiT and Generic Grid Services
Middleware (Re:GRIDiT) Layer
Computing Resources
Storage Resources
Base Layer
High Bandwidth Network

Figure 4.1: System Architectural Levels.

namic applications for users that require resources and new functionality on demand.

4.2 Middleware Level System Model

In the following we present a formal description of our system model, and introduce the following concepts: A *site* is the basic operational entity in the system. We assume a fail-stop model for site failures. A *Grid system* is a collection of sites equipped with a Grid middleware. We assume there is an underlying network protocol that can be used to send messages reliably from one site to the other with known bounded delay.

4.2.1 Data Model

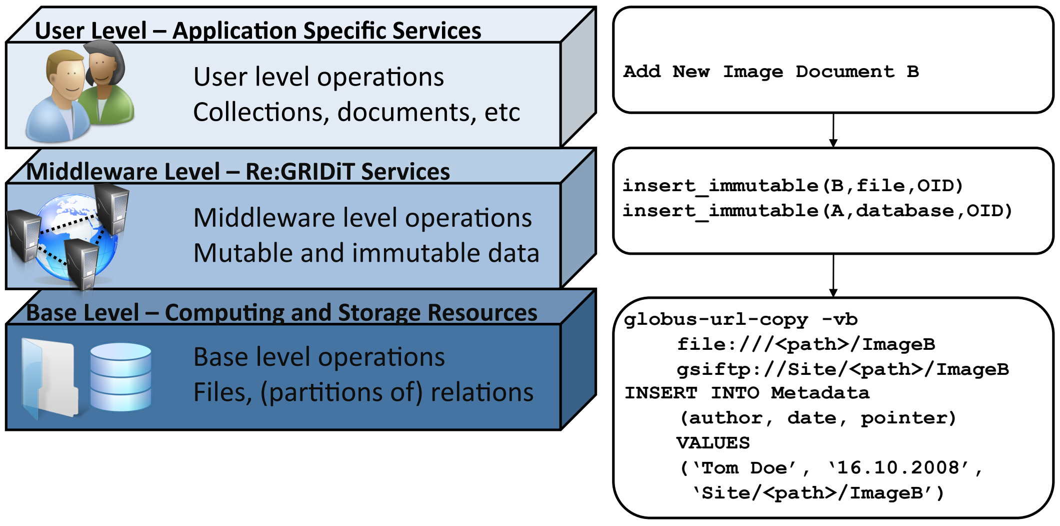
Throughout this thesis the term *data object* is used to denote a container for an information object, or for the document to information object relation, or for the collection set, or for various kinds of feature indexes used to support text, image, or audio/video search support. Approaches where the indexes used in the transformation between user objects and physical objects are considered as physical objects as well, and are later on processed, already exist (search engines such as FAST follow such an approach [75]).

In the following, we present the three layers in more detail, describing the data objects and operations available at each level. This description is meant to facilitate the understanding of how data and operations are mapped between different levels. A more detailed description of our operations model will follow.

At the **base layer**, we assume a Grid network which provides a general distributed computing environment in which each site is able to communicate to (all) sites in the network in a reliable manner. This layer is an abstraction from basic Grid storage facilities and provides a basic infrastructure consisting of (any) relational database and a file management system. Data objects can be represented at the base level as files stored on the local file system, database relations, or partitions of a relation in the local database management system (similar to the approach presented in [40]). Our protocol relies on the fact that local, base level operations on objects stored in the database are executed as local database transactions, and base level operations on objects stored on the local file system can take advantage of Grid file management functionality, such as GridFTP [97].

We envisage Re:GRIDiT at the **middleware layer** as being part of the overall Grid middleware. It is present on each site and provides transparent support for replication and distributed transaction management. At this level, the sites hold data objects, which are replicated in the network, and operations through which the data objects can be accessed. We assume that one physical object can reside only on one site, but could be replicated on several sites. At this level data objects are classified into *mutable* and *immutable* data objects. Immutable data objects are created only once and kept until deleted. They cannot be modified, but may have several versions (for example, satellite pictures, X-ray pictures of a patient's internal organs, etc.) Mutable data objects can be modified and therefore do not have versions (for example, environmental reports, interpretations of X-ray or ultrasound pictures of human internal organs i.e., annotations which summarize what the pictures indicate). At the same time, a set of semantically rich operations is available at the middleware level for each type of data objects.

At the **user layer**, we consider three types of data objects: collections, documents and information objects. Each collection consists of one or more documents, shared between collections. A document may belong to more than one collection. Each document consists of one or more information objects. Collections, documents and information objects are transparently mapped to data objects at the middleware level. Similarly, the operations exposed to the


User Level – Application Specific Services
User level operations
Collections, documents, etc
Middleware Level – Re:GRIDiT Services
Middleware level operations
Mutable and immutable data
Base Level – Computing and Storage Resources
Base level operations
Files, (partitions of) relations

```
Add New Image Document B
```

```
insert_immutable(B,file,OID)
insert_immutable(A,database,OID)
```

```
globus-url-copy -vb
    file:///<path>/ImageB
    gsiftp://Site/<path>/ImageB
INSERT INTO Metadata
    (author, date, pointer)
    VALUES
    ('Tom Doe', '16.10.2008',
     'Site/<path>/ImageB')
```

Figure 4.2: Example of User Transaction Mapping between Levels.

user which allow the creation, access, and manipulation of collections, documents, and information objects are automatically and transparently mapped to operations acting on middleware data objects. This mapping determines a unique OID for the data objects to be accessed, and these OIDs are referred in all operations.

Our data model classification into mutable and immutable data objects comes from the valid observation that there are typically two kinds of objects representing scientific data: data content and metadata objects. Generally, data content objects are immutable. Only new versions of these objects can be added. Metadata objects, on the other hand, are generally mutable. It is important to observe that collections and documents are always mutable, since we always can either add/delete a document to/from a collection and by the same token, we always can add/delete an information object to/from a document. Following the same reasoning, indexes are also mutable and are used to define dependencies between objects. Consequently data objects are never treated in isolation and consistency is always maintained, since for example adding a new object to a collection will always mean that the index is recalculated, even for data with lower freshness levels. In short, immutable objects at the middleware level can be only information objects. All other types of objects are considered to be mutable.

Figure 4.2 gives an example of how the mapping between the different layers is performed. If we come back to the earth observation scenario presented in Chapter 2, we can imagine users, at the user level working with operations that allow them to manipulate satellite images or collections of such images. A scientist working on the oil slick distribution problem and which requires a new image for his environmental report would start at the user level the following transaction: "Add New Image Document B". This transaction will be automatically transformed at the middleware level into "insert image B on the file system" and "insert metadata A (associated to B) into the database". These operations will be in turn mapped to base level operations that any Grid infrastructure is able to handle (such as GridFTP copy or SQL insert). The image associated to document B will be stored on the file system, while metadata associated to the image and a pointer to the location of the file will be stored in the database. IBM DataLinks[1] is another example of a successful application that allows the storage of large files such as documents, images, video in the file system in order to take advantage of file-system capabilities, while at the same time coordinating the management and access to these files and their contents with associated data stored in a relational database management system.

In a similar fashion, the creation of a collection of images at the user level will be mapped to the following sequence of operations at the middleware level: "insert images on the file system", "insert metadata (associated to the images) into the database", "insert information regarding the containment of the images in the collection into the database". These operations will result at the base level in the insertion of several tuples in a database relation (so that the membership in a collection is finally mapped to a horizontal partition of this table) and the copying of images via GridFTP to the designated sites.

Re:GRIDiT is capable of supporting arbitrary physical layouts ranging from full replication at the granularity of complete databases to partial replication. The partial replication scheme does not require all data objects to be replicated on each site. Smaller subsets of the entire set of data objects are replicated on the different sites in the system. This way, two sites may contain different partitions which may even be overlapping. Throughout the thesis, we refer to sites in the network as update and read-only sites. The assumption behind is always that a site can be an update site for a certain data objects and read-only site for others. In practice, however, for performance reasons, update sites should be considered updateable for all data objects existing on the sites. As subsequent chapters will reveal, our replication protocol makes no assumptions regarding the replication scheme.

4.2.2 Operations Model

With respect to the operations model, we assume an approach in which each site s_j offers a set of semantically rich operations [181] on data objects $Op^{s_j} = \{op_1, op_2, \ldots, op_n\}$ (see Figure 4.3). These operations can be invoked

[1]http://www.almaden.ibm.com/resources/datalinks.pdf

within transactions using the interface of that site. The operations at the middleware level can be classified according to three criteria. According to the first criterion, the operations are classified depending on the type of the data object they refer to. Consequently, we distinguish between mutable and immutable operations.

We assume the following middleware level operations for immutable data objects: $insert$, $delete$, and $read$. As already mentioned, immutable data objects cannot be updated. In turn, one can create a new version of an immutable data object. In more detail, the operations on immutable data objects are:

- $insert\_immutable(object, type, OID)$, where the parameters represent the following: $object$ – the data object to be inserted, $type$ – the type of storage to be chosen, either a database relation, partition of a relation or a file, and OID – the unique OID of the object. If the object does not exist it will be created with version 0. Otherwise a new version of the data object is created and the version number is returned.

- $delete(OID)$, deletes an object with the specified OID.

- $read(OID)$, reads an object with the specified OID. This operation always reads the last version of the object.

- $read\_version(OID, number)$, reads a particular version of the object with the specified OID.

Mutable objects can be $inserted$, $replaced$, and $read$. We assume that a $replace$ operation always acts on the freshest version of a data object. The semantics of the middleware level operations for mutable data objects are presented below:

- $insert\_mutable(object, type, OID)$, where the parameters represent the following: $object$ – the data object to be inserted, $type$ – the type of storage to be chosen, either a database relation or partition of a relation or a file, and OID – the unique OID of the object.

- $replace(object, OID)$, replaces an object with the specified OID. The execution of this operation has as consequence the removal of the previous object from all sites where it resides, and the updated data object is copied on all sites.

- $read(OID)$, reads an object with the specified OID. This operation always reads the last version of the object.

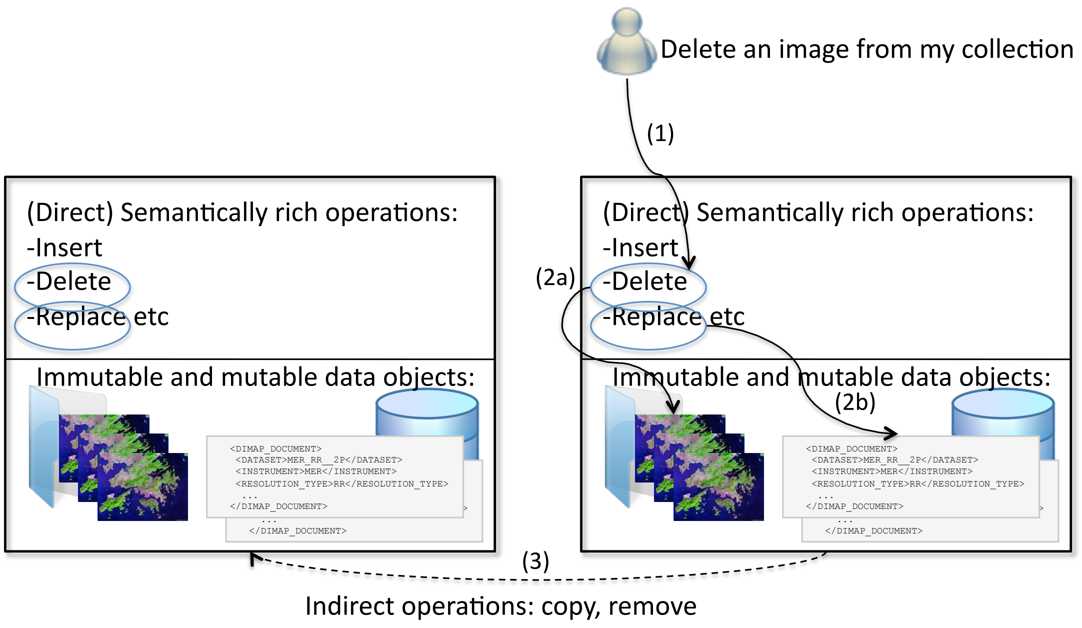

Delete an image from my collection
(1)
(Direct) Semantically rich operations:
-Insert
-Delete
-Replace etc
Immutable and mutable data objects:
(2a)
(2b)
(3)
Indirect operations: copy, remove

Figure 4.3: Data and Operations at the Middleware Level.

Our second classification criterion distinguishes between operations according to how these operations are initiated. All the operations presented so far on both mutable and immutable data object are the result of the translation of user operations into operations that the system has to deal with at the middleware level. We call these operations in our model *direct operations*. In addition, we assume two additional operations that are not available at the user level, operations which are used by the system in order to support replication (*indirect operations*). For immutable objects, these indirect operations are:

- $\mathrm{copy}(\mathrm{OID}, \mathrm{number}, \mathrm{destination})$, copies a particular version of the object with the specified OID on a destination site. It is triggered automatically whenever a new data object or a newer version of a data object has been created (by using an $\mathrm{insert\_immutable}$).

- $\mathrm{remove}(\mathrm{OID}, \mathrm{number}, \mathrm{destination})$, removes a particular version of the object with the specified OID from a destination site. It is executed whenever a data object has been $\mathrm{deleted}$ from a site.

The indirect operations for mutable objects are:

- $\mathrm{copy}(\mathrm{OID}, \mathrm{destination})$, copies the object with the specified OID on a destination site.

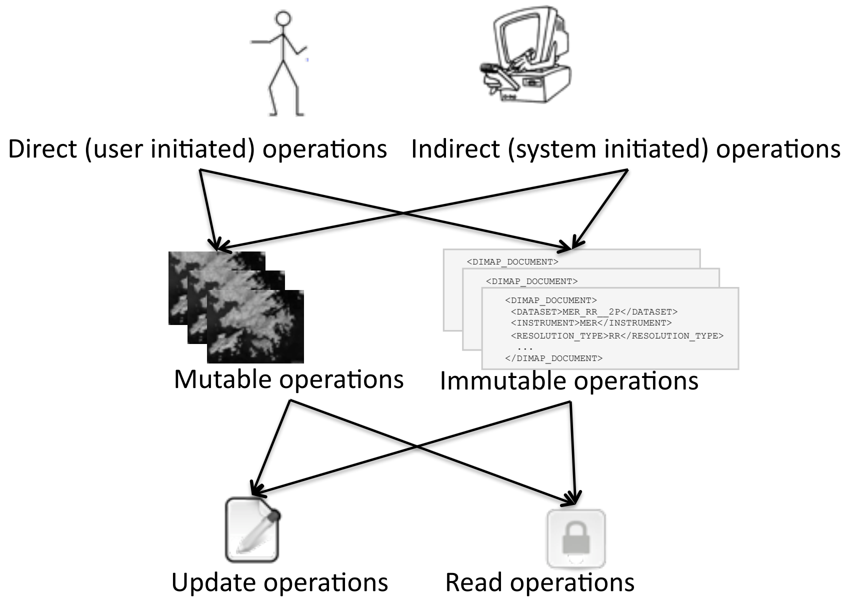

Direct (user initiated) operations
Indirect (system initiated) operations
<DIMAP_DOCUMENT>
<DATASET>MER_RR__2P</DATASET>
<INSTRUMENT>MER</INSTRUMENT>
<RESOLUTION_TYPE>RR</RESOLUTION_TYPE>
</DIMAP_DOCUMENT>
Mutable operations
Immutable operations
Update operations
Read operations

Figure 4.4: Classification of Operations at the Middleware Level.

- $remove(\text{OID}, destination)$, removes the object with the specified OID from a destination site.

Unlike a *delete* operation, which deletes the data object from every site in the network, $remove$ only "deletes" the data object from the site where the operation was submitted. These operations have been introduced in order to support replication, therefore they act on single replicas, whereas $insert, replace, delete$ act on global objects and lead, as a consequence, to changes being made to all replicas in the system.

Consider for example again a scientist in our earth observation scenario who is a working in a project involving oil spill distribution studies on a certain region. In order to produce new results he needs to eliminate an image from his collection, as new, more accurate images become available. His transaction at the user level (labeled with (1) in Figure 4.3) is mapped at the middleware level to a sequence of direct operations: $delete$ of that particular image (labeled (2a)) and the $replace$ of the collection index to preserve the integrity of the collection (labeled (2b)). At the same time, our protocol will ensure the consistency of the data at the update sites and via indirect operations apply the same changes at all the other replicas in the system (labeled with (3)).

The third criterion for the classification of operations refers to how the operations access data. We consider the following update operations (shortly referred to as updates): $insert\_mutable$, $insert\_immutable$, $delete$ and $replace$. We consider the following read operations: $read$ object (which reads of the freshest copy or the last version number of a specified object) and $read\_version$ (which reads a version of the object or a copy of the object with a particular freshness value).

The complete classification of the operations is presented in Figure 4.4.

4.2.3 Network Topology

As previously introduced, a site is the basic operational entity in the system. A site contains data objects that are replicated in the system, and operations by means of which the data objects can be accessed. Depending on the type of access that they allow to their data, the sites are divided into two classes (similar to the approaches presented in [10, 157]): we classify the sites as *update* sites and *read-only* sites. Read-only sites only allow read access to data. Updates occur on the update sites and are propagated to other update sites and finally to the read-only sites. We assume an eager replication among update sites and in addition we apply lazy replication mechanisms between update and read-only sites that take into account different levels of freshness (using the definition introduced in [157]). In order to facilitate the propagation of updates from the update sites to the read-only sites and to better support users queries with different freshness levels, we define an *1-to-n* relationship between the two types of sites; an update site can have any number of read-only children (to which updates are propagated). Read-only sites can further propagate their changes to other read-only sites, thus maintaining different versions and different levels of freshness of data in the system. As a failure handling mechanism only, read-only sites can be shared with other update sites. In order to ease the distinction between the different types of sites, we introduce the following naming convention, which defines a tree structure. Since there are many update sites per data object, the result is a forest of trees in the system (see Figure 4.5):

- *Level 1* (root) sites are all the update sites, where the data are synchronized (eager update everywhere replication). More importantly, we assume that there is some form of clock synchronization between these sites, needed for the calculation of the freshness levels.

- *Level 2* sites are read-only sites which are kept as up-to-date as possible. We define an *1-to-n* relationship between Level 1 and Level 2 sites.

- *Level 3* (optional) sites are read-only sites where copies of data are not frequently updated. We define an *1-to-n* relationship between Level 2 and Level 3 sites. Level 3 sites are optional, but provide a definite advantage if the number of read-only sites in the system is considerably

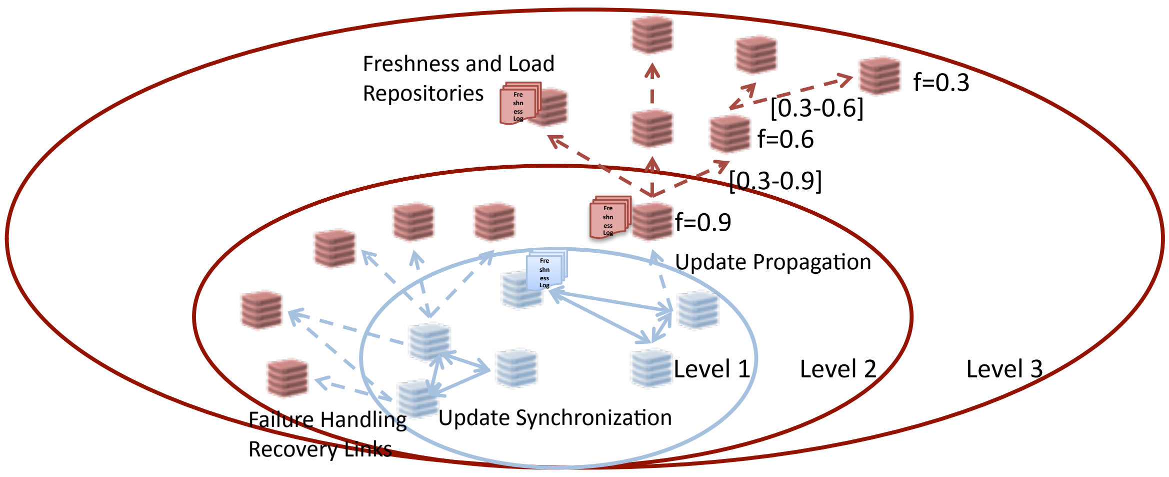


Figure 4.5: Tree Topology of the System at the Middleware Level.

bigger than the number of update sites (as a trade-off between consistency and performance). Level 3 read-only sites may themselves be hierarchically structured, i.e., form tree structures of different depths, and may propagate in turn updates to their own children or may re-route request to their parents if they are not able to service request locally.

The following update operations are available through the interface of Level 1 update sites: *insert_mutable*, *insert_immutable*, *delete* and *replace*. We consider the following read operations which can be invoked on all sites: *read* (which reads of the freshest copy or the last version number of a specified object) and *read_version* (which reads a version of the object or a copy of the object with a particular freshness value).

By using the tree topology, we can ensure that the freshness level for a particular data object is monotonically decreasing along the tree hierarchy. This feature will allow us to make an efficient routing of the queries along the tree structure.

4.2.4 The Initial Configuration

In this section, we show how the tree is initially constructed. In order to create a tree, we require a minimum number of initial update replicas (at least three) that are used to bootstrap the tree and a set of fixed IP addresses. These initial IP addresses point to (some) sites in the system equipped with the Re:GRIDiT middleware, which in turn are able to (transitively, following the tree hierarchy) locate (at least) one update site.

Any new site that wishes to join the replication scheme can do so, on the lowest level in the tree hierarchy. By using one of the IP addresses in the fixed set the new replica can be directed to the lowest site in the tree hierarchy. For example, in the initial phase of a tree constructed with three update sites, any new replica will become a Level 2 read-only site. After this, new sites can enter the scheme by becoming read-only sites, children of an existing site on the current lowest level in the tree hierarchy. Continuous propagation transactions ensure that the new replica will eventually hold a copy of the data object(s) of interest with a certain freshness level. From this point on, the dynamic load balancing between the replicas presented in Chapter 6 and the dynamic tree changes induced by refresh transactions presented in Chapter 7 ensure that a site can dynamically move up or down in the tree hierarchy and be promoted to an update site or again demoted to a read-only site based on access patterns that change over time.

4.2.5 Distributed System Repositories

Each site uses a set of tools to obtain a (typically partial and sometimes even out-dated) information regarding the state of the system and take replication decisions. We introduce the following components to facilitate the scheduling of read-only transactions in the Grid and the replica management decision. These components are not centrally materialized in the system, and contain global information that is distributed and replicated to all the sites in the system:

Replica catalog: used to determine the currently available replicas in the network, as follows: Level 1 (update) replicas contain distributed replicated information about the other update sites in the network and about their corresponding tree. In order to minimize the exchange of information and yet ensure that any update site can eventually reach any other update site (for example, for update transactions that include two or more data objects), update replicas need be aware of only one update (Level 1) site for each data object in the network. For failure handling purposes, more than one Level 1 site per data object can be managed in the local replica repository. We take advantage of the continuous propagation transactions in order to replicate this information within each tree. Each update (Level 1) replica is aware of its own Level 2 read-only replicas, where it propagates update changes, in order to maintain a certain level of freshness/staleness in the network. Each read-only replica is aware of its parent site in the tree and its subordinate read-only sites (if any).

Freshness repository: used to collect the freshness of data objects periodically or on-demand. In this thesis, the term freshness is used to emphasize the divergence of a replica from the up-to-date copy. Consequently update sites will always have the highest freshness, while the freshness of the read-only sites will measure the staleness of their data. The sites at the leaves of the tree have the stalest data in the network. Irrespective of their level, sites are aware of the freshness levels of the sites to which they (transitively) propagate changes. As observed from Figure 4.5 sites in the system are aware of the freshness intervals that can be provided by their subordinates in the tree.

Propagation queues: are used to enqueue changes that need to be applied to subordinate sites in the tree. These changes are bulked into propagation transactions and applied to the sites whenever possible. The propagation transactions execute changes from the local propagation queues in order. These queues are continuously updated as new updates arrive at the update sites. These updates are collected together into packages of fixed size which are later on propagated to other sites.

Load repository: used to determine an approximate load information concerning the sites. This information can then be used to balance the load while routing read-only transactions to the sites or for the replica selection algorithm. Update sites periodically receive information regarding the load levels of other update sites and their subordinate children. Read-only sites are only aware of their own load levels. In order to improve efficiency and not to increase the message overhead this information is exchanged together with replica synchronization request. This information needs to be exchanged more frequently while there is a replica synchronization process in place, while the exchange is not needed during a site's idle time, when the load is unlikely to vary.

4.2.6 Transaction Model

Based on the concepts introduced in Chapter 3, we define the following notions that apply to our transaction model. A transaction T_i is a pair $(O_{T_i}, \ll_{T_i})$, where O_{T_i} is the set of operations to be invoked and $\ll_{T_i}$ is a partial order defined over O_{T_i}.

The notion of a schedule is fundamental for defining a criterion for correct concurrent executions of transactions although no central scheduler exists and thus no complete schedule is materialized in the system. A *schedule* S is a pair $(O_S, \ll_S)$ with O_S being the operation invocations from all transactions in the system and $\ll_S$ the order between these invocations.

The notion of conflict is defined based on the commutativity behavior of operation invocations (semantically rich operations). Dependencies between transactions in a schedule occur when there is at least a pair of service invocations in conflict.

Let op_1 and op_2 be two operation invocations. Then, op_1 and op_2 commute if for any pair of sequences α, β of operation invocations the return values are the same in $\langle\ \alpha\ op_1\ op_2\ \beta\ \rangle$ and $\langle\ \alpha\ op_2\ op_1\ \beta\ \rangle$. Otherwise, the operation invocations are in conflict, i.e., $(op_1, op_2) \in$ CON with CON $\subseteq \{O^{T_1}, ..., O^{T_n}\} \times \{O^{T_1}, ...O^{T_n}\}$ being the conflict relation. A transaction T_i depends on a transaction T_k in a schedule S if there exists a pair of conflicting service invocations $op_i \in O^{T_i}$ and $op_k \in O^{T_k}$ such that op_i occurs before op_k in S, i.e., $(op_i, op_k) \in$ CON and $op_i <_S op_k$.

Each transaction T owns a local serialization graph SG_T which comprises the conflicts in which T is involved. Essentially, the graph contains at least all conflicts that cause T to be dependent from other transactions. This partial knowledge is sufficient for a transaction to be able to decide whether it is allowed to commit. Note that a transaction can only commit after all transactions on which it depends have committed.

Let SG_T be the local serialization graph of transaction T. The nodes that can (transitively) be reached from the node T via a directed path represent the (transitively) *post-ordered* ($POST(T)$) transactions of T. Analogously, the nodes for which there is a directed path to the node are called the (transitively) *pre-ordered* ($PRE(T)$) transactions of T.

Note that a transaction may not always succeed due to several failure reasons. To satisfy the demand for an atomic execution, the transaction must compensate the effects of all the operations invoked prior to the failure [181]. This compensation is performed by invoking semantically inverse operations in reverse order. It results in a straightforward manner that the following pairs of operations compensate each other: $(insert\_immutable, delete)$, $(insert\_mutable, replace)$ and $(copy, remove)$.

We assume that each site owns a conflict matrix that defines the conflicts among the operations offered by this site. Using the conflict matrix, a site is able to detect conflicts between operation invocations of different transactions. The conflict matrices for update transactions are presented in Tables 1 and 2.

As it can be seen, not all operations are available to all data object types. Depending on whether the data object is mutable or immutable a different set of operations may be used to manipulate it. Table 1 describes the conflict matrix for immutable data objects. Table 2 describes the conflict matrix for mutable data objects. As we can observe from the shadowed right lower corner of both conflict matrices, this distinction between mutable and immutable data objects strongly influences the conflict definition when it comes to operations that manipulate data: *insert, replace, delete*. The lower right part of both

	Insert	Delete	Read_version	Read	Copy	Remove
Insert	+	-	+	-	-	-
Delete	-	+	+	-	-	+
Read_version	+	+	+	+	-	-
Read	-	-	+	+	-	-
Copy	-	-	-	-	+	-
Remove	-	+	-	-	-	+

Table 1: Conflict Matrix for Immutable Data Objects

	Insert	Replace	Read	Copy	Remove
Insert	+	+	+	+	+
Replace	+	-	-	-	-
Read	+	-	+	-	-
Copy	+	-	-	+	-
Remove	+	-	-	-	+

Table 2: Conflict Matrix for Mutable Data Objects

tables (concerning *read, copy* and *remove* operations, shadowed in both tables) remains unchanged, since replication is independent of data object types. It is important to notice here that the shadowed right lower corner of the matrices are in fact the conflict matrices for the read-only sites, where only read transactions are allowed.

We define the commutativity of two operations as follows: by changing the order in which the operations are invoked, the system cannot detect any changes at the end of the operations invocations (a formal definition can be found in [181]). Moreover, we assume that operations are atomic and leave the system in an unchanged state if they fail.

With respect to transactions submitted by clients we distinguish between *read-only transactions* and *update transactions* [10]. A read-only transaction only consists of read operations. Read operations within a single read-only transaction may run at several different read-only sites. An update transaction contains at least one update operation. Decoupled *refresh transactions* propagate updates through the system, on-demand, in order to bring the read-only replicas to the freshness level specified by the read-only transaction. A refresh transaction aggregates one or several propagation transactions into a single bulk transaction and is comprised of update operations. A refresh transaction is processed when a read-only transaction requests a version that is younger than the version stored at the read-only site. *Propagation transactions* are performed during the idle time of a site in order to propagate the changes present in the local propagation queues to the read-only replicas. Therefore, propagation transactions are continuously scheduled as long as

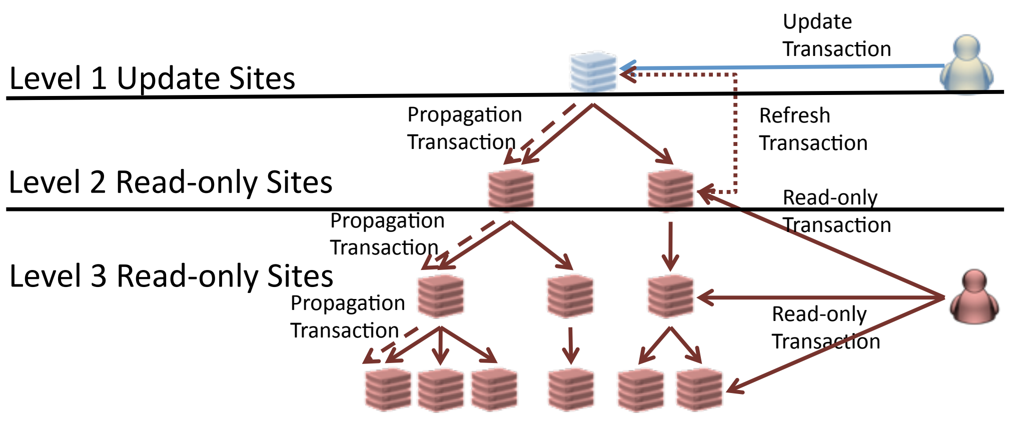


Figure 4.6: Propagation of Various Transaction Types along the Hierarchical Network Structure.

there is no running read or refresh transaction. Copies at the read-only sites are kept as up-to-date as possible such that the work of refresh transactions (whenever needed) is reduced and the overall performances is increased. The different transaction types are presented in Figure 4.6.

4.2.7 Failure Handling

Various kinds of failures can occur in distributed systems. In the following we briefly present these types of failures. The failure handling mechanisms that are implemented to prevent failures during different phases of the protocol are discussed in more detail in subsequent chapters of this thesis.

An isolation failure is materialized by a cycle in the serialization graph. Our protocol ensures that all local serialization graphs are acyclic at commit time and no transaction which has active dependencies in its serialization graph is allowed to commit, which guarantees a serializable schedule.

A communication failure is considered to have occurred when: (i.) A message gets corrupted during communication between two sites; (ii.) A message is lost due to malfunctioning of a network link or (iii.) Two sites cannot communicate due to unavailability of a network path.

A site failure may be due to system failure. We assume that a site fails by stopping. This means that either it is operating correctly or not operating at all, it never operates incorrectly. We assume such failures to be temporary and that upon recovery the internal state is still available.

4.2.8 Logging and Recovery

Logging and recovery are concepts that all commercial database management systems must support to ensure the various ACID properties of transactions. We use a similar approach and allow operations to be logged (or recorded) at the physical level and these logs record what happens in the storage structures of the database or the file system. Each change to the storage structures has its own log entry, which describes the structure being changed and what the change was. This is done in such a way that the change can be replayed or reversed, if necessary. Our model assumes the presence of local logs on each site, where every operation invocation is recorded. The local logs are also used to determine conflicts between transactions and to determine the actions to be taken in case an operation needs to be compensated. The local logs have been implemented in Re:GRIDiT as tables in the local databases. Each row contains operation invocations that may have to be rolled back, and potential conflicts that these invocations have caused, according to the conflict matrix. When a transaction commits, its records are no longer needed for recovery purposes, but they are kept together with records of refresh and propagation transactions in order to ensure that updates are not lost.

4.3 Summary

In this chapter we lay the basis for the Re:GRIDiT family of protocols and describe a system model for our data Grid replication system. We propose a layered architecture and a middleware-based approach in which clients interact through the middleware rather than directly with the individual sites. Furthermore we formally describe the data model that can be supported by the Re:GRIDiT protocols and the components which will be required in order to seamlessly support replication. Re:GRIDiT is capable of supporting arbitrary physical layouts ranging from full replication at the granularity of complete databases to partial replication. Since Re:GRIDiT allows partitioning of data, a data object can be either a file or database relation or a partition of a relation.

We divide the sites in the network between update and read-only sites, serving update and read-only transactions. This clear separation between sites with different functionality will allow us combine different replication mechanisms and provide efficient and reliable access to data.

Last but not least, we introduce various components which are required by our completely decentralized approach. By means of several distributed and replication repositories, information of interest (such the local load, freshness levels of different data objects) is propagated through the system in order to facilitate the routing of queries and the selection of best replicas for particular workload situations.

5 Coordination of Distributed Update Transactions on Replicated Data

In the previous chapter we have presented our system model and the components which are required at the middleware level in order to facilitate the building blocks for the Re:GRIDiT family of protocols. In this chapter we concentrate on the first Re:GRIDiT protocol, namely Re:SYNCiT which coordinates the distribution of updates to replicated data in a completely decentralized way. The Re:SYNCiT protocol is based on optimistic transaction management techniques and relies on communication between transactions and sites to ensure global correctness. Re:SYNCiT extends and generalizes the DSGT protocol presented in [100, 101]; its novel features, which make it uniquely flexible, are: transparent support for data replication at Grid scale and seamless integration of a generic data model that satisfies the needs of a large variety of Grid user communities. The Re:SYNCiT protocol provides reliable support for replication management and builds upon the following observations: (i) dependencies between transactions are managed by the transactions themselves, (ii) global correctness can be achieved even in the absence of a global coordinator and with incomplete knowledge by communication between transactions and sites, (iii) replication and data versioning are handled by the sites, completely transparent to the transactions and the users.

5.1 Problem Statement

Newly emerged eScience domains pose interesting challenges that have still remained unanswered. Efficient solutions that are capable of solving all the problems of use case scenarios such as the ones presented in Chapter 2 do not exist yet. In terms of data management, the Grid allows keeping a large number of replicas of data objects to provide a high degree of availability, reliability and performance. Replication management in the Grid needs to be able to deal with a potentially large number of updateable replicas per data object. As the survey presented in Chapter 3 shows, a replication management protocol that provides a high degree of availability, reliability and performance and that supports concurrent updates to different replicas, while at the same time taking into account the particular characteristics of the Grid (most importantly the absence of a global coordinator) also does not exist yet. Despite the considerable work done in the context of distributed transaction management and replication management, there is no protocol which can be seamlessly applied to a data Grid environment without impacting correctness and/or overall performance. In order to develop such management schemes one needs to resolve data availability, data consistency, and data versioning issues for data Grid environments. We address all these problems in the Re:SYNCiT protocol, presented in the following.

5.2 Properties of Re:SYNCiT

The main idea of Re:SYNCiT is that global correctness can be ensured by communication between transactions and sites, thus eliminating the need for a global coordinator as in traditional approaches such as two-phase commit (2PC). Each transaction maintains a local serialization graph which is exchanged with other transactions in the network whenever conflicts occur. A transaction invokes operations on data objects optimistically, without requesting any locks. The sites where it has invoked operations reply with a list of local conflicts, based on the local conflict matrix. With this information, the transaction updates its serialization graph and informs its pre-ordered transactions. Since we assume a replicated system model, each direct operation (except for *read* operations) on a data object at each site needs to be propagated to all the update sites in the system that contain a replica of that particular data object. In our protocol we delegate this task to the sites. The sites trigger the indirect update operations within the boundaries of the originating transaction, but completely transparent to the originating direct update transaction. This means that each update transaction is automatically extended by the corresponding indirect operations. For the originating transaction it is not important to know immediately if one of the indirect operations has caused a conflict on a remote site. It suffices that it will know about this conflict at commit time. Before commit, the transactions are informed of conflicts caused by their indirect operations, and update their serialization graphs accordingly. Since Re:SYNCiT relies on serialization graph testing protocols and the invocation of operations is done optimistically, it cannot prohibit cycles in the serialization graph, but it guarantees that the committed projection remains acyclic.

We assume conflicts to be rather infrequent, but they need to be handled properly in order to guarantee globally serializable executions. We distinguish between: local conflicts, as determined by the sites using their local conflict matrices (caused by direct operation invocations), and remote conflicts, appeared as a result of replication propagation (caused remotely by indirect operation invocations).

5.3 The Re:SYNCiT Protocol

The Re:SYNCiT protocol requires communication between transactions and sites in order to ensure global correctness and support replication. Therefore the algorithm we propose is decoupled into two parts, one running on each site, one for each transaction (run by the middleware which executes the transaction).

The part of the protocol that runs for each transaction consists of three phases (execution, validation, and commit). From a transaction's point of view, replication, data versioning, and data separation into mutable and immutable data objects are supported in the validation phase, where the transactions are made aware of the remote conflicts they may have caused (Algorithm 1 and 2).

Another part of the protocol runs on the sites. At every site, the site protocol is able to detect and handle local conflicts on the spot, based on the local conflict matrix. In addition, remote conflicts need to be detected correctly. The site protocol performs actions as responses to messages received (Algorithms 3, 4 and 5).

Re:SYNCiT is furthermore capable of supporting arbitrary physical layouts ranging from full replication at the granularity of complete databases to partial replication. The partial replication scheme does not require all data objects to be replicated on each site. Smaller subsets of the entire set of data objects are replicated on the different sites in the system and different sites may contain different overlapping partitions.

5.3.1 Transaction States

Algorithms 3 to 5 define the part of the protocol that runs on each site. The part of the protocol that runs on each transaction consists of two threads (see

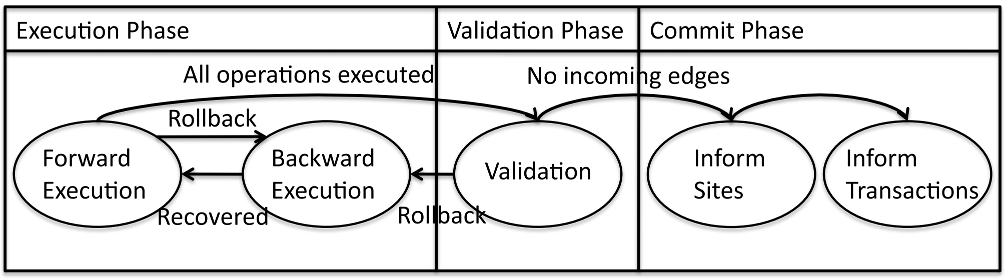


Figure 5.1: Possible Transaction States and State Transitions.

Algorithms 1 and 2). The main execution thread is always in one of the following states, which correspond to the states in which a transaction can exist during its lifetime (see Figure 5.1).

Forward Execution: The transaction invokes operations on sites according to interface of that site.

Backward Execution: The transaction rolls back partially by invoking compensation operations in the corresponding inverse order.

Validation: The transaction waits until the corresponding node in the local serialization graph has no more incoming edges. Only in this case the transaction is allowed to commit.

Inform Sites: The transaction informs all sites on which it has invoked operations about its commit. Therefore, the transaction not only gets the information from these sites about its post-ordered transactions, but it also prevents that the site will return a conflict where this transaction is involved in.

Inform Transactions: Finally, the transaction informs all its post-ordered transactions about its commit.

The following state transitions can take place:

Forward Execution $\rightarrow$ **Validation**: If the transaction has executed successfully all specified operations, it changes to the validation state.

Validation $\rightarrow$ **Backward Execution**: If the transaction detects that it is involved in a cycle and that it has been chosen as victim (it will rollback), it changes to the backward execution state.

Forward Execution $\rightarrow$ **Backward Execution**: This transition happens when the transaction must roll back due to the rollback of a pre-ordered transaction or when the local serialization graph contains a cycle and the transaction is the victim.

Backward Execution $\rightarrow$ **Forward Execution**: As soon as all required operation invocations are compensated, the transaction changes to forward execution again.

Validation $\rightarrow$ **Inform Sites**: This transition occurs when the validation yields that the local serialization graph contains an incoming edge from an active transaction.

Inform Sites $\rightarrow$ **Inform Transactions**: This transition occurs as soon as all corresponding sites have responded to the transaction.

Algorithm 2 also describes the part of the protocol that runs on the transactions to detect cycles in the local serialization graph.

In the following subsection, we show the phases of the Re:SYNCiT protocol and how Re:SYNCiT ensures global correctness.

5.3.2 Protocol Phases

The part of the protocol that runs for each transaction consists of three phases (execution, validation, and commit), detailed in the following:

1. *The execution phase:* The transaction invokes its operations according to the operation interface provided by each site. Sites execute operations in an optimistic manner without requesting any locks. The transaction receives, upon the completion of the operation, the result of the invocation and a list of local conflicts, if any (i.e., these are conflicts with other operations which have been previously invoked by other transactions on the same site). The transaction uses this information to update its local serialization graph, and if there are any changes the graph is propagated to all its pre-ordered transactions (i.e., to all transactions for which a site has reported a conflict after the invocation of an operation).

2. *The validation phase:* When a transaction has finished executing all its specified operations, it enters in the validation phase. However, up to this point a transaction only knows about the local conflicts it has generated on the sites where it executed operations (either on the basis of the information directly returned from the site after an operation invocation or by exchanging serialization graphs with other transactions). Therefore it will contact all the sites where it executed operations to request updated information about possible remote conflicts that indirect update operations executed by the sites on its behalf have generated on remote sites. If new dependencies have emerged in the local serialization graph, the transaction will propagate these changes to all its pre-ordered transactions. The transaction can now validate if its serialization graph is acyclic. If there are no incoming edges, i.e., if the transaction does not depend on any active transaction, it will proceed to the next phase. Otherwise, it will wait until the corresponding active transactions have committed, i.e., until there are no incoming edges in the serialization

Algorithm 1 Transaction Protocol

```
1:  $O_T^* := [o_1, ... , o_n]$ // sequence of direct operations to be invoked by T;
2:  // we assume any $o_i$ belongs to the set: {insert, read, replace, delete};
3:  $S_T^* := \{\}$ // sites on which T invoked operations;
4:  $SG_T := \{\}$ // local copy of the serialization graph;
5:
6:  Main Thread:
7:  // 1. Execution Phase:
8:  for all $o_i \in O_T^*$ do
9:      //invoke operations and update the serialization graph
10:     invoke $o_i$ to an appropriate site $s$ and add $s$ to $S_T^*$;
11:     wait for reply from $s$;
12:     // site will eventually send return values of the operation and a list of
        local conflicts
13:     // and update the graph with new conflicts
14:     update $SG_T$ based on reply information;
15:     if new dependencies emerged then
16:         //propagate changes to pre-ordered transactions
17:         propagate $SG_T$ to $PRE(T)$;
18:     end if
19: end for
20: // 2. Validation Phase:
21: for all $s \in S_T^*$ do
22:     request update from $s$;
23:     wait for reply from p;
24:     // site will eventually send a list of all remote conflicts
25:     update $SG_T$ based on reply information;
26:     if new dependencies emerged then
27:         //propagate changes to pre-ordered transactions
28:         propagate $SG_T$ to $PRE(T)$;
29:     end if
30: end for
31: // this step is performed by the parallel thread Serialization Graph Up-
    date Thread
32: wait until $SG_T$ does not contain any incoming edges for $T$;
33: // 3. Commit Phase:
34: for all $s \in S_T^*$ do
35:     send "Commit" to $p$;
36:     update $SG_T$ based on reply information;
37: end for
38: mark T as "committed" in $SG_T$;
39: propagate updated $SG_T$ to $POST(T)$;
40: terminate
```

Algorithm 2 Transaction Protocol (continued)

```
1: Serialization Graph Update Thread:
2: while true do
3:   wait for message with $SG_T$ as an update serialization graph or $SG_T$
     locally updated;
4:   if message arrived then
5:     update $SG_T$
6:   end if
7:   if $SG_T$ changed then
8:     send $SG_T$ to $PRE(T)$
9:     if $SG_T$ is cyclic and T is victim then
10:      abort;
11:    end if
12:  end if
13: end while
```

Algorithm 3 Site Protocol

```
1: $T_{conflicts} = \emptyset$;
2: while true do
3:   wait for next message $m$;
4:   if $m$ contains message: directly execute $o_i$ from transaction T then
5:     execute $o_i$;
6:     schedule indirect operation invocation of $o_i$;
7:     for all e $\in$ Log do
8:       if $e.T' \neq$ T and $(e.o, o_i) \in$ CON then
9:         $T_{conflicts} = T_{conflicts} \cup$ T;
10:      end if
11:    end for
12:    add $(o_i$, T) to Log;
13:    return $T_{conflicts}$;
14:  else if $m$ contains message: indirectly execute $o_i$ from T then
15:    execute $o_i$;
16:    for all entries e $\in$ Log do
17:      if $e.T' \neq$ T and $(e.o, o_i) \in$ CON then
18:        $T_{conflicts} = T_{conflicts} \cup$ T;
19:      end if
20:    end for
21:    add $(o_i$, T) to Log;
22:    return $T_{conflicts}$;
23:  end if
24: end while
```

Algorithm 4 Site Protocol (continued)

```
while true do
  wait for next message $m$;
  if $m$ contains message: update conflicts for T then
    $S^* := [s_1, \dots, s_n]$ // sites with replicas of data objects accessed by T;
    while $s_i \in S^*$ has not replied with remote conflicts do
      wait;
    end while
    for all $e \in$ Log and $e.T = T_{updating}$ do
      for all $e' \in$ Log with $e' \neq e$ do
        if $e'.T \neq T_{updating}$ and $(e.o, e'.o) \in$ CON then
          $T_{conflicts} = T_{conflicts} \cup e.T$;
        end if
      end for
    end for
  else if $m$ contains message: transaction T will commit then
    $T_{post} = \emptyset$
    for all $e' \in$ Log with $e' > e$ do
      if $e'.T \neq T_{committing}$ and $(e.o, e'.o) \in$ CON then
        $POST(T) = POST(T) \cup e'.T$;
      end if
    end for
    for all $e \in$ Log do
      if $e.T = T_{committing}$ then
        remove e from Log
      end if
    end for
  end if
end while
```

graph. If the transaction detects that it is involved in a cycle and the victim selection algorithm has chosen it as victim, it will abort.

3. *The commit phase:* A transaction T in the commit phase has sufficient knowledge to deduce from its own local serialization graph that it is safe to commit. This is the case when it does not depend on any active transaction, i.e., when there is no incoming edge to T in the serialization graph of T. The transaction commits and informs all sites where it has executed operations about its commit. The sites compile a list of all post-ordered transactions which need to be informed about the commit. This is necessary because the post-ordered transaction might already be in the validation phase and waiting to commit themselves.

Algorithm 5 Site Protocol (continued)

```
1: Schedule Indirect Operations Invocations:
2: $S^*$ := [$s_1$, ... , $s_n$] // sites which contains replicas of data objects accessed by T;
3: while true do
4:    wait for next message $m$
5:    if $m$ contains message: schedule indirect operation invocation of $o_i$ of transaction T then
6:       for all $s_i \in S^*$ do
7:          send message to $s_i$: indirectly execute $o_i$;
8:       end for
9:    else if reply received from $s_i$ (as a log entry $e_i$) then
10:      for all $e \in Log$ with $e \neq e_i$ do
11:         if $e.T \neq T_i$ and $(e.o, e_i.o) \in CON$ then
12:            $T_{conflicts} = T_{conflicts} \cup T_i$;
13:         end if
14:      end for
15:      return $T_{conflicts}$;
16:   end if
17: end while
```

Another part of the protocol runs on the sites. At every site, the site protocol is able to detect and handle local conflicts on the spot. In addition, it also needs the the ability to correctly detect remote conflicts. The site protocol performs the following:

1. *Direct operation execution.* In case of a direct operation invocation from a transaction T, the site s executes the operation optimistically, without requesting any locks. The site checks for local conflicts, if any, in the local log. The operation result together with a list of conflicts, if any, is returned to T. The operation invocation is then stored in the local log.

2. *Indirect operation invocation.* If the object which is being accessed is replicated, site s which has received the direct operation executes the operation remotely and in parallel on all the sites where a replica of the same data object resides, on behalf of transaction T. These invocations are called indirect operation invocations.

3. *Indirect operation execution.* In case a site s' receives an indirect operation invocation from a remote site s, on behalf of a transaction T that has accessed on the remote site s a replicated data object, the operation is executed optimistically, and the local conflicts are sent back to the re-

mote site s which has triggered the indirect operation invocation. The operation invocation is stored in the local log of s'.

4. *Management of remote conflicts.* In case of a reply from a remote site s' as a consequence of an indirect operation invocation initiated by the local site s, site s stores the list of conflicts, if any, in the local log.

5. *Update of local and remote conflicts.* In case of an update message from a transaction T (as part of the transaction's validation phase), site s will provide the list of all local and remote conflicts. Each site knows exactly what data object have been accessed by each transaction and how many sites hold replicas of the same data objects in the network from the replica catalog. The site will return the whole list to the transaction only when all the replica sites have replied. This information is used by the transaction to update their local serialization graph with remote conflict information.

6. *Propagation of commit information.* In case of a commit message from a transaction T, the site s will provide the list of all post-ordered transactions. All of them will be informed by T of its commit, and update their local serialization graph accordingly. At this point the information about the committed transaction is removed from the local log.

Summing up, transactions invoke operations without determining on the spot the correctness of the serialization graph. Nevertheless, a transaction T_i in the validation step will commit if and only if the following condition holds: $\nexists\ T_k : T_i \neq T_k$ and $T_i \in \mathsf{POST}(T_k)$. In other words, a transaction will only then commit when it does not depend on any active transactions.

Knowing when a transaction is allowed to commit is not enough, the system must also be able to detect and resolve cyclic dependency situations, i.e., situations in which cyclic dependent transactions prohibit each other to commit. None of the involved transactions will be able to commit and this situation will not change without intervention. Moreover, since in this kind of situations there are usually two or more transactions executed on different sites involved, neither a single transaction, nor a single site can detect this situation using their partial local knowledge.

There are several approaches available known from the area of distributed deadlock detection which could be applied to solve this problem. We assume in our protocol the following approach, inspired by path-pushing approaches such as the one presented in [129], which covers the following aspects:

1. If a transaction causes a new conflict, it will propagate the changes to its pre-ordered transactions based on the updated serialization graph.

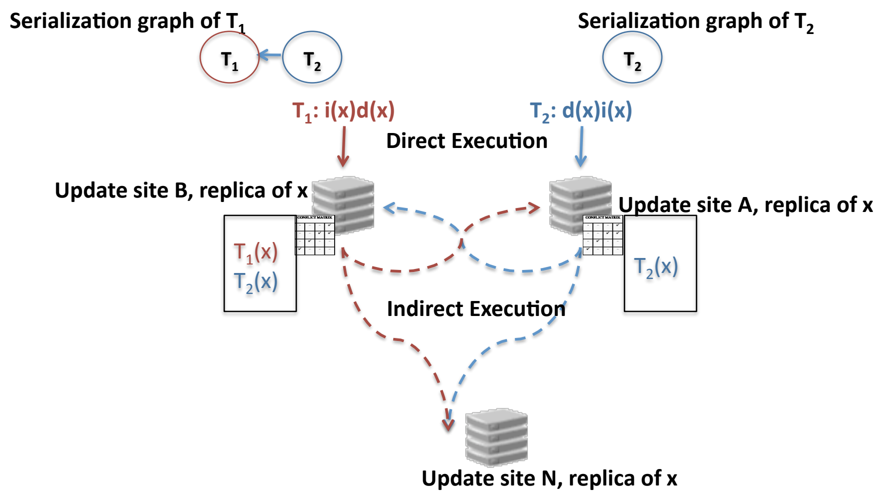

Serialization graph of T1
T1
T2
Serialization graph of T2
T2
T1: i(x)d(x)
T2: d(x)i(x)
Direct Execution
Update site B, replica of x
Update site A, replica of x
T1(x)
T2(x)
T2(x)
Indirect Execution
Update site N, replica of x

Figure 5.2: The Re:SYNCiT Transaction Protocol - The Execution Phase (operation 1).

2. If a transaction receives such changes from another transaction, it will update its own serialization graph and propagate it to its pre-ordered transactions.

3. If a transaction detects a cycle and the victim selection strategy has selected itself as a victim (using an algorithm as presented in [192]), it aborts.

We employ partial rollback techniques in order to avoid cascading aborts similar to the approach presented in [101].

Let's consider the following very simple example of two conflicting transactions. The example is purposely chosen to produce conflicts in the transactions' serialization graphs. This example serves the purpose of showing how our protocol handles cyclic dependency situations which might result in an inconsistent state. We take advantage of this example and explain the various phases of a transaction's lifetime. In real-life use case scenarios, transactions would consist of more operations and their rollback and recovery would become more complex.

Assume the following two transactions, $T_1 :< insert(x)delete(x) >$ and $T_2 :< delete(x)insert(x) >$. The concurrent execution of both transactions on different update sites would produce different results. On one site, $< insert(x)delete(x) >$ would have as consequence the fact that x is no longer present on the site, while on the other site $< delete(x)insert(x) >$ would first try to delete object x and then insert it, resulting as a consequence in the fact that x exists on the site.

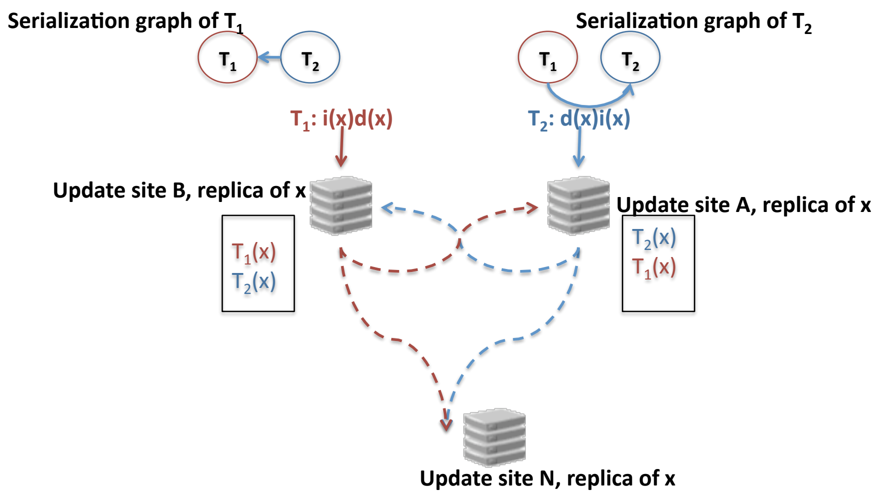

Serialization graph of T1
T1
T2
Serialization graph of T2
T1
T2
T1: i(x)d(x)
T2: d(x)i(x)
Update site B, replica of x
T1(x)
T2(x)
Update site A, replica of x
T2(x)
T1(x)
Update site N, replica of x

Figure 5.3: The Re:SYNCiT Transaction Protocol - The Execution Phase (operation 2).

Let's see how this situation is handled in a completely distributed way by the Re:SYNCiT protocol. In the first phase, transactions execute operations optimistically without assuming any locks. Assume that transaction T_2 is directed by the middleware to update site A, which contains an up-to-date replica of x. This invocation is recorded in the site's local log. Assume that transaction T_1 is directed by the middleware to update site B, which also contains a replica of x. At the same time, the $delete(x)$ from site B is propagated to site A. According the site A's conflict matrix, the two operations conflict. They are both recorded in the site's local log and this local conflict is returned to transaction T_1. T_1 accordingly updates its serialization graph. The $insert(x)$ from site A is also propagated to site B (and of course at the same time to all other update sites that contain a replica of x), but since transaction T_2 is unaware of this operation invocation, its serialization graph does not reflect it (yet). The execution of the first operations of the transactions in first phase of the protocol is illustrated in Figure 5.2. Once the two transactions have finished executing all their operations, we have the following situation reflected in the two transactions' serialization graph: T_1 depending on T_2 and T_2 depending on T_1. However, this information is distributed between the two transactions and up to this point none of them has the complete information to make an informed decision about whether or not they can safely commit (see Figure 5.3).

For this reason, once they have finished invoking all operations the transactions enter a validation phase when they go back to all sites where they have previously invoked operations and request a list of remote conflicts they may have indirectly caused. As already explained before, remote conflicts are

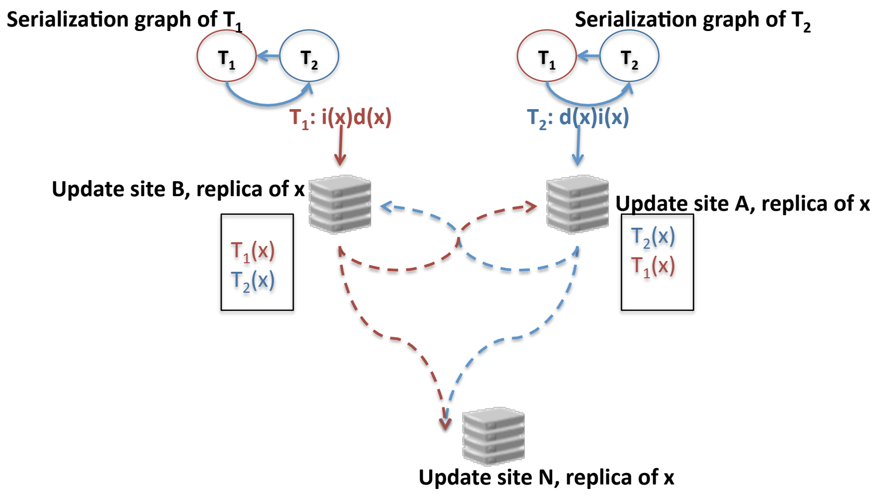

Serialization graph of T1
T1
T2
Serialization graph of T2
T1
T2
T1: i(x)d(x)
T2: d(x)i(x)
Update site B, replica of x
Update site A, replica of x
T1(x)
T2(x)
T2(x)
T1(x)
Update site N, replica of x

Figure 5.4: The Re:SYNCiT Transaction Protocol - The Validation Phase.

caused by the sites while propagating changes on behalf of a transaction, in order to support replication. With this information the transactions properly update their serialization graphs, and exchange relevant pieces of information with other depending transaction in their graph. This way, the two transactions T_1 and T_2 become aware of the cyclic dependency that they are involved in (case illustrated in Figure 5.4).

Once a cycle has been detected by the cycle detection mechanism, a victim is chosen to break the cycle. In case of failures, locking-based protocols roll back one transaction completely. Other active transactions are not affected by the rollback. Since optimistic protocols do not use a locking mechanism, the rollback of a transaction (which has been involved in a cycle and has been chosen as victim by the victim selection strategy) may lead to the rollback of at least all the transactions involved in the cycle (the problem of cascading aborts). In order to reduce the costs of rollback, our protocol uses partial rollbacks, where a transaction does not rollback completely in case of isolation failures, but only up to a point in time until the incoming and outgoing edges in the serialization graph contributing to the cycle have disappeared. The victim is completely compensated. Obviously this approach minimizes the effects of cascading aborts but the choice of the proper victim selection can become a parameter for further tuning. There are several victim choice strategies available, such as *the youngest transaction* (which implies less operation invocations to compensate), *most number of incoming edges* (which would remove as many edges as possible), *least number of outgoing edges* (which would again imply less operation invocations to compensate), etc. The implications of choosing one strategy over the other have already been discussed in the

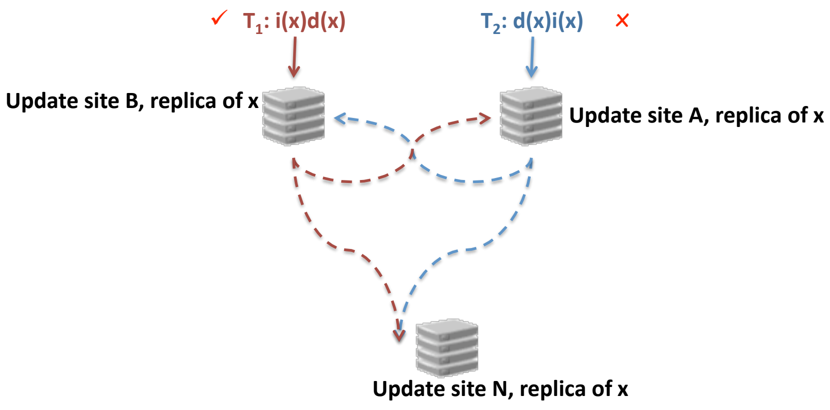

✓ T1: i(x)d(x)
T2: d(x)i(x) ×
Update site B, replica of x
Update site A, replica of x
Update site N, replica of x

Figure 5.5: The Re:SYNCiT Transaction Protocol - The Commit Phase.

literature [100, 192]. In the experimental evaluation of this protocol we have used the youngest transaction strategy.

Assume in our case that victim selection strategy has chosen T_2 as a victim (case illustrated in Figure 5.5). In this case, transaction T_1 will roll back up to a point in time when the cycle has disappeared, while T_2 will be totally compensated. According to the conflict matrix, its operations will be semantically undone. T_1 will resume forward execution when the cycle has disappeared and it will hopefully commit.

5.3.3 Failure Handling

Various kinds of failures can occur in distributed systems. Briefly summarized, our protocol is able to support short-term disconnects by using timeout intervals. Long-term disconnects are detected by the expiration of the timeout period and are treated by removing the replica from the catalog and rolling back the transactions active on that replica site.

An isolation failure is materialized by a cycle in the serialization graph. The Re:SYNCiT protocol ensures that all local serialization graphs are acyclic at commit time and no transaction which has active dependencies in its serialization graph is allowed to commit, which guarantees a serializable schedule. Moreover, each transaction (eventually) detects a cycle it is involved in. A formal proof of correctness follows.

A communication failure is considered to have occurred when: (i.) A message gets corrupted during communication between two sites; (ii.) A message is lost due to malfunctioning of a network link or (iii.) Two sites cannot communicate due to unavailability of a network path.

In the first two cases, we rely on network protocols to provide reliability. We assume that the network protocol implementation takes measures against message corruption by using appropriate error correction coding techniques. We also assume that the transport protocols and the networking infrastructure takes care of masking malfunctioning of network links by re-routing and re-transmitting packets. The last case might occur due to network partition. A network partition can occur if a combination of sites and network links between the sites fail. Such failures can be avoided by designing networks with redundant communication paths. We take it for granted, for the time being, that when such failures do occur, the necessary actions are taken by the underlying messaging system infrastructure to mask against such failures.

A site failure may be due to system failure. We assume that a site fails by stopping. This means that either it is operating correctly or not operating at all, it never operates incorrectly. We assume such failures to be temporary and that upon recovery the internal state is still available. As transactions and sites talk to each other by sending messages in the system, a failure can be detected if the sender does not receive the expected reply (an acknowledgment message) within a timeout period. The failure to receive an expected message could be either due to a communication link failure or a site failure for the receiver. We also assume that the timeout period is calculated by taking into account events such as intermittent overloading of the communication network or the load on the partner from which the message is expected. If such a site failure is detected (failure to acknowledge the receipt of a message), the system will not send new requests or new direct operations to the failed sites and indirect operations belonging to transactions started before the failure are queued until the recovery of the failed site. If the site failed to recover within the timeout period, a consensus is reached between the remaining sites and effects of the indirect operations propagated by the failed site are undone. The site is removed from the replica catalog and will be considered a new replica upon recovery.

5.3.4 Protocol Optimization

We propose the following *Early Cycle Detection* optimization strategies for our protocol. This protocol optimization can increase performance by reducing the waiting time of a transaction in the validation phase. In our protocol we assume that remote conflicts are detected the earliest in the validation phase. When a transaction has finished execution and is ready to commit it goes back to all the sites where it has previously invoked operations to request updated information about the remote conflicts that indirect updates have caused on its behalf. We propose the following protocol optimization: whenever a site receives a reply from a remote site with a list of conflicts that the indirect updates has caused on the remote site, the local site contacts the direct update transaction and inform it about the conflicts. In this way, the waiting time needed in the commit phase is reduced, since the transaction has to wait for less updates from remote sites and communication overhead can be traded for a reduction of the waiting time in the validation phase. Communication overhead can be further more reduced is sites wait for a certain number of conflicts before reporting them to the transaction, thus reducing the number of times a transaction is contacted by a site.

5.4 Proof of Correctness

In traditional systems, the correctness of schedules is guaranteed by the presence of a global coordinator. Our correctness criterion is inspired by the unified theory of concurrency control and recovery [181]. The basic requirement of the theory is the availability of inverse operations that semantically undo the effects of the already executed operations. Therefore we have to ensure that the serialization graph is acyclic when the effects of the compensation operations are removed. The serialization graph is a directed graph where the nodes are represented by the transactions, the directed edges correspond to the conflicts between transactions and the direction of the edges indicates the order of the conflicting operation invocations. Since our protocol relies on serialization graph testing protocols and the invocation of operations is done optimistically, our protocol cannot prohibit cycles in the serialization graph, but it guarantees that the committed projection remains acyclic.

We avoid incorrect schedules by guaranteeing (i) recoverability and (ii) serializability. The latter means that no cycles in the committed projection [28] of a schedule may appear, whereas the first aspect prohibits a transaction to commit while it depends on an uncommitted transaction. Moreover, given the absence of a global coordinator, and in the presence of replication we can prove that our protocol guarantees that the serialization order is the same at all sites.

Theorem 5.1 *Algorithms 1 to 5 together generate only executions for which the following holds: all local serialization graphs are acyclic at commit time and no transaction which has active dependencies in its serialization graph is allowed to commit.* □

Proof. Assume a local schedule S in which a transaction T_c has committed although it is dependent on at least one active transaction. Then there must be a transaction T_p such that the conflict $T_p \rightarrow T_c$ holds.

If T_c has committed, it must have successfully passed the validation phase in Algorithm 1, lines 23-38. In this case, the serialization graph of T_c cannot have contained a conflict leading to the edge $T_p \rightarrow T_c$ at that time. There are two possibilities when this can happen:

1. T_c never knew about this conflict.
2. T_c knew about the conflict, but removed (invalidated) it before.

Both cases however lead to a contradiction to our assumption.

1. Since the edge $T_p \rightarrow T_c$ ends at T_c, T_p appeared as a consequence of an operation invocation of T_c (in case of a direct operation) or on behalf of T_c (by an indirect operation). This site would have returned $T_p \rightarrow T_c$ and T_c inserted it in its graph (Algorithm 1, lines 13-17 or 25-28). Hence, T_c would have known about the conflict $T_p \rightarrow T_c$ in line 31. Consequently, T_c would not have proceeded and therefore not committed. This however contradicts our assumption.
2. The removal (invalidation) of the conflict $T_p \rightarrow T_c$ with the active transaction T_p can theoretically only happen in the following situations:
   - T_c marks T_p as committed. This only appears as a consequence of T_p marking itself as committed (line 43-44) and spreading this information. Since we assumed that T_p is still active, this cannot be the case.

   Since this cannot lead to a loss of this information, T_c would never be able to validate correctly and therefore cannot commit. This contradicts our assumption that T_c commits.

Hence, our initial assumption cannot be correct which implies the correctness of the theorem. □

Theorem 5.2 *Algorithms 1 to 5 ensure that each transaction (eventually) detects a cycle it is involved in.* □

Proof. Let $T_1, \ldots T_n$ be involved in a cycle $T_1 \rightarrow T_2 \rightarrow \ldots T_i \rightarrow T_{i+1} \ldots \rightarrow T_n \rightarrow T_1$. Thus, proving the theorem requires to proof that each of these transactions gets the information of each $T_i \rightarrow T_{i+1}$ with $n + 1 \equiv 1$. Without loss of generality, we show that T_i receives the information about each conflict $T_i \rightarrow T_{i+1}$. We prove this in two steps:

1. T_{i+1} sends a message to T_i when it causes the conflict $T_i \rightarrow$ Ti+1 .
2. If T_{i+1} receives the first time a message containing the conflict $T_j \rightarrow T_{j+1}$ for any j = i, then it sends this information to T_i.

We start with statement (1):

1. A service invocation of T_{i+1} causes the conflict $T_i \rightarrow T_{i+1}$ by invoking an operation on a site S_k. Thus, S_k returns this conflict to T_{i+1}. T_{i+1} inserts this conflict into its graph (Algorithm 4.2.1, lines 13-17, 25-28). Afterwards, T_{i+1} sends this information to T_i (line 20, 31). Therefore, statement (1) holds.

2. In this case, T_{i+1} receives a message about the conflict $T_j \rightarrow T_{j+1}$. Here, we distinguish two sub-cases:

   (a) $SG_{T_{i+1}}$ T_{i+1} already contains $T_i \rightarrow T_{i+1}$, when the message arrives (line 51). Then, T_{i+1} inserts the conflict $T_i \rightarrow T_{i+1}$ into its local graph (line 52). Afterwards, T_{i+1} sends its updated graph to its pre-ordered transactions (line 55). Since T_i is the recipient of this message, this information finally arrives there.

   (b) $SG_{T_{i+1}}$ T_{i+1} does not contain the conflict $T_i \rightarrow T_{i+1}$. When the message arrives (line 51), T_{i+1} inserts this conflict into its local graph (line 52). Because T_i is not known to be pre-ordered with respect to T_{i+1} , T_i is not a receiver when T_{i+1} forwards the updated graph (line 55). However, according to our protocol, T_{i+1} will receive the conflict $T_i \rightarrow T_{i+1}$ from the corresponding site. Then, T_{i+1} inserts this conflict into the local graph (lines 13-17, 25-28). Now T_i is a pre-ordered transaction from the point of view of T_{i+1}. Therefore, the graph which already contains $T_j \rightarrow T_{j+1}$ is sent to T_i (line 20, 31). Thus, also T_i receives the information on this conflict.

Since cases (a) and (b) are supported, statement (2) also holds such that cycles are eventually detected by the associated transactions. □

Theorem 5.3 *Algorithms 1 to 5 together generate only executions for which the following holds: the serialization order of conflicts on both local and remote sites is the same at commit time.* □

Proof. Assume an execution in which a transaction T_c is in conflict with T_p. Assume further that on site S_1, the following serialization order is reported $T_c \rightarrow T_p$ and at another site, S_2, the serialization order derived from the local order of conflicts is $T_p \rightarrow T_c$. This situation could occur in the following two cases:

1. S_1 is the site where T_c has invoked a direct operation which led to the conflict, and that S_2 is the site where the corresponding indirect operation has been executed on behalf of T_c.

2. T_c has invoked a conflict operation at S_2 and the corresponding indirect operation has been executed on behalf of T_c at S_1.

In both cases, it is irrelevant whether the operation of T_p involved in the conflict is a direct or an indirect one at S_1 and S_2.

Consider the commit of T_c. In the validation phase (Algorithm 1, lines 23-38), T_c updates its local serialization graph by contacting the site where it has executed the direct operation. In return, T_c, receives information about the remote conflicts that have occurred.

In the first case, this returns the serialization order $T_p \rightarrow T_c$ which has been reported to S_1 by S_2. This means, that T_c is dependent on T_p and not allowed to commit (Algorithm 1 line 32). Moreover, it will inform T_p about this order (Algorithm 1, line 36). At the latest when T_p is about to commit, it will also update its local serialization graph by requesting its remote conflicts and receives the information $T_c \rightarrow T_p$. Thus, T_p will be able to detect a cycle in its local serialization graph and trigger its resolution (line 56). Finally, the cyclic conflict will be removed by aborting at least one of the two transactions involved in the cycle.

In the second case, in order to update the local serialization graph during the validation phase (lines 23-38), T_c has to contact S_2 and receives in return the serialization order $T_c \rightarrow T_p$ which has occurred at S_1. Since it has already received the serialization order $T_p \rightarrow T_c$ during the execution phase (line 51) after the direct operation has been executed on S_2, T_c can immediately detect the cycle in its local serialization graph and trigger its resolution. Finally, the cyclic conflict will be removed by aborting at least one of the two transactions involved in the cycle.

In both cases, this leads to a contradiction of the initial assumption of different serialization orders at the two sites. □

Theorem 5.4 *Algorithms 1 to 5 together generate only executions for which the following holds: all global serialization graphs are acyclic at commit time and no transaction which has active dependencies in its serialization graph is allowed to commit.* □

Proof. Assume a global schedule S in which a transaction T_c has committed although it is dependent on at least one active transaction. Then there must be a transaction T_p such that the conflict $T_p \rightarrow T_c$ holds.

Since according to Theorem 5.1, all local serialization graphs contain no conflicts at commit time, we must assume that the conflict $T_p \rightarrow T_c$ is remote. However, according to Theorem 5.2 all local and remote serialization orders are the same.

This invalidates our initial assumption of an acyclic global serialization graph. □

Theorems 5.1 to 5.4 together prove that the Re:SYNCiT protocol ensures global correctness by guaranteeing both recoverability and serializability in a distributed environment and in the absence of a global coordinator.

5.5 Comparison to Existing Approaches

We are faced nowadays with the need for more and more complex computations on huge amounts of data that take place in Grid infrastructures. There exists naturally the need to ensure a correct execution and guarantee the success of such computations, by enforcing transactional guarantees on their executions.

A decentralized serialization graph testing protocol that ensures concurrency control and recovery in peer-to-peer environments has been proposed in [101, 177]. Although the paper is proposing a model based on known techniques, such as serialization graph testing and partial roll backs, it combines old techniques for a new purpose, presuming globally correct execution of concurrent transactions based on the local knowledge of the transactions. A similar approach has been recently adopted for peer-to-peer transaction management [17]. However, the proposed protocols do not handle replication and data versioning.

As applications scale to take advantage of Grid resources, their size and complexity will increase dramatically. Grid computing is a set of standards and technologies that academics, researchers, and scientists around the world are developing to help organizations take collective advantage of improvements in microprocessor speeds, optical communications, raw storage capacity, and the Internet. This new emerging technology has been gaining a lot of attention from the birth. However, very little effort has been spent so far in the area of Grid transactions.

The architecture GridTP, developed at the Shanghai Jiang Tong University [147], is based on the Open Grid Services Architecture (OGSA) platform and the X/Open DTP model, providing a consistent and effective way to make available autonomously managed databases in the Grid. This means that a two-phase commit protocol is used to atomically commit distributed Grid transactions.

5.6 Summary

We have presented a new protocol for the synchronization of updates between the update sites of a data Grid which combines an optimistic concurrency control with eager replication mechanisms. The central idea is to enable transactions and sites with the ability to communicate with each other in order to guarantee global correctness in the absence of a global coordinator. Data replication and data separation between mutable and immutable data types are transparently handled by the sites. To the best of our knowledge currently available protocols neither deal with replicated data in Grid environments, nor allow such a complex model which handles data versions or different types of data. Yet these are vital requirements for a large range of applications.

In this chapter we proposed the first protocol that solves these problems: Re:SYNCiT hides the presence of replicas to the applications, takes into account the special characteristics of data in the Grid such as version support, distinction between mutable and immutable objects, and provides provably correct transactional execution guarantees without any global component.

6

Load-Aware Dynamic Replication in a Data Grid

The previous chapter has introduced the first pillar of our Re:GRIDiT approach to data Grid replication. In this chapter, we introduce our second pillar, the Re:LOADiT approach to dynamic distributed replica management in data Grid systems, which takes into account several important aspects described below. First, replicas are placed according to an algorithm that ensures load balancing on replica sites and minimizes response times. Another important issue is choosing the optimal number and location of replicas. The denser the distribution of replicas, the shorter the distance a client site needs to travel to access a data copy. Nevertheless, since maintaining multiple updateable copies of data is expensive, the number of updateable replicas needs to be bounded. Optimizing access cost of data requests and reducing the cost of replication are two conflicting goals and finding a good balance between them is a challenging task. We propose efficient algorithms for selecting optimal locations for placing the replicas so that the load among the replicas is balanced. Given the data usage from each user site and the maximum load of each replica, our algorithm efficiently minimizes the number of replicas required, reducing the number of unnecessary replicas. This approach to dynamic replica management is embedded into the Re:SYNCiT replica update protocol which guarantees consistent interactions in a data Grid and takes into account concurrent updates in the absence of any global component.

6.1 Problem Statement

Novel data-intensive applications are increasingly popular in eScience. In these applications, vast amounts of data are generated by specialized instruments and need to be collaboratively accessed, processed and analyzed by a large number of scientists around the world. Examples of such applications have already been introduced in Chapter 2. The size of data required by these applications will easily grow up to Petabytes. The nearly unlimited storage capabilities of data Grids allow these data to be replicated at different sites in order to provide a high degree of availability. Dealing with such large amounts of replicated data, geographically distributed across several sites, poses many challenges, including data placement, versioning, data freshness and data consistency. At the same time the particular constraints of a data Grid have to be taken into account: in contrast to distributed data management in a cluster, data Grids need to deal with the heterogeneity of sites and the absence of any global component that would become a performance bottleneck or a single point of failure. In particular when data are concurrently updated by several clients, correctness and consistency need to be guaranteed in a completely distributed way. The same is true for data placement when data access patterns change over time.

6.2 Load Metrics

One of the main objectives of dynamic replica placement in distributed systems is load balancing. In order to reach this objective we need to first define the concept of "load". In practical scenarios, load metrics may reflect multiple components, such as CPU load, storage utilization, disk usage or bandwidth consumption, etc. or any combination of them. In this work, we abstract from the notion of load by defining a load function $load \in [0, 1]$, where the maximum value, in the case of CPU load, for instance, would correspond to 100 % load.

We assume that an individual site can measure its total load. This can be done by keeping track of resource consumption (CPU time, IO operations, access count rate etc.). In order to ensure an accurate measurement, the load collection is averaged over a certain time window and consequently the decisions are made based on load trends rather than punctual values.

We define the mean load as $\overline{load(s)} = \frac{\sum_{i=0}^{n} load(s_i)}{n}$ where n represents the total number of load calculations within the given time window. Choosing the right size for the time window in practical scenarios is a question of the nature of the applications, and in general some monitoring is required in order to fine tune this parameter. Furthermore, other definitions such as running mean (for instance) could be used here.

In the following we provide a categorization of the notion of load, based on various load situations that can occur during the lifetime of a site s. The rationale behind our classification is the following: earth observation application scenarios such as the one presented in Chapter 2 have shown the need for a dynamic replica management. Based on that use case, the occurrence of a significantly important event (for example, a tanker accident followed by an oil spill) implies an increase in user requests (i.e., higher load) and it should seamlessly lead to an increase in the number of replicas for data objects of importance (for example images and information about the region affected by the oil spill). At the same time increasing the number of updateable replicas per data object in an unlimited way may have significant drawbacks on the overall system performance. Also, when the sites are idle, the number of updateable replicas should be reduced to a minimum as they are no longer needed.

Based on these observations we define four load intervals, during the lifetime of a site: an *underload* interval, when the sites are mostly idle, therefore a high number of replicas is of no use in the absence of clients requests. A *lowload* interval is considered to be the normal functioning load level of site, during which the replication mechanism will maintain the status-quo. Crossing that level, we would then reach a *heavyload* interval, during which an increasing number of user requests for certain data objects should lead to a dynamic increase in the number of replicas for those objects so that the system is capable to serve requests in a timely manner. The last and highest level of load, the *overload* level is the case in which a site is overcome by user requests such that it is no longer capable of functioning efficiently. User requests that are directed to an update site might be of two types: update or read operations. Our middleware will efficiently route read-only transactions to read-only sites, nevertheless an update transaction may contain arbitrarily many read operations. We argue that in this context, read operations contribute to a site's local load, whereas updates contribute to the global system load, since updates are synchronously replicated. The overload situation is consequently a special case in which the global load (not just the local load) needs to be taken into account. If a single site (or a minority of them) is in an overload situation it can safely re-route read requests to less loaded sites and thus alleviate its local load. However, if this is majority situation, more drastic measures are required (releasing a replica might in the long run help as it will reduce the communication overhead for replica synchronization).

These intervals are described in a more formal way as follows:

An **overload** situation can occur if $\overline{\mathtt{load}(s)} \in (\alpha_{\mathtt{overload}}, 1]$,
where $\alpha_{\mathtt{overload}} \in [0; 1]$.

A **heavyload** situation can occur if $\overline{\mathtt{load}(s)} \in (\alpha_{\mathtt{heavyload}}, \alpha_{\mathtt{overload}}]$,
where $\alpha_{\mathtt{overload}} \in [0; 1]$ and $\alpha_{\mathtt{heavyload}} \in [0; 1]$.

A **lowload** situation can occur if $\overline{\mathtt{load}(s)} \in (\alpha_{\mathtt{underload}}, \alpha_{\mathtt{heavyload}}]$,
where $\alpha_{\mathtt{heavyload}} \in [0; 1]$ and $\alpha_{\mathtt{underload}} \in [0; 1]$.

An **underload** situation can occur if $\overline{\mathtt{load}(s)} \in [0, \alpha_{\mathtt{underload}}]$,
where $\alpha_{\mathtt{underload}} \in [0; 1]$.

The parameters $\alpha_{\mathtt{underload}}$, $\alpha_{\mathtt{heavyload}}$ and $\alpha_{\mathtt{overload}}$ are application dependent and obey the following relation:
$0 \leq \alpha_{\mathtt{underload}} \leq \alpha_{\mathtt{heavyload}} \leq \alpha_{\mathtt{overload}} \leq 1$.

We extend the above classification to a system-wide level (i.e., a *system overload* can occur if $\frac{\sum_{j=0}^{N} \overline{\mathtt{load}(s_j)}}{N} \geq \alpha_{\mathtt{overload}}$, where $\overline{\mathtt{load}(s)}$ is the average load on a site s and N represents the total number of sites in the system) and define two additional situations that can be observed at a system level:

A **majority overload** situation can occur if a majority of sites (of at least $\frac{N}{2} + 1$) is in an overload situation. The remaining sites must be at least in a heavyload situation.

A **minority overload** situation can occur if a minority of sites (of at most $\frac{N}{2} - 1$) is in an overload situation. The remaining sites must be at least in a heavyload situation.

As mentioned before, the definition of the load levels in a practical scenario is of major importance since, crossing the boundaries of a certain level will trigger a certain action in the dynamic replication protocol.

6.3 Best Replica

The efficiency of the replication system depends on the criteria used for the selection of replica sites. In our protocol, we choose the notion of host proximity as one of the criteria for replica selection. The proper definition of this metric impacts both the response time perceived by the user and the overhead involved in measuring the distance. In order to define the "closest" replica, the metrics may include response time, latency, ping round-trip time, network bandwidth, number of hops or geographic proximity. Since most of the above metrics are dynamic, replica selection algorithms such as [182] typically rely on estimating the current value of the metric using samples collected in the past. We abstract from the above notions by defining a *closeness function*, in which the notion of closeness can reflect any of the metrics above, and is defined as $\mathtt{close}(s, c) \in [0, 1]$, where 0 is the absolute minimum value for this metric (for example, in the case of geographic proximity, 0 would correspond to the site s that is geographically farthest away from the client c and 1 to the site s that is geographically closest to the client c).

The second criterion for the replica selection is based on the notion of "freshness". We use in our approach a *freshness function*, $\mathtt{fresh}(d, s) \in [0, 1]$, (as defined in [157]) that reflects how much a data object d on a site s has deviated from the up-to-date version. The most recent data has a freshness of one, while a freshness of zero intuitively represents infinitely outdated data (data object is not present on that site). For read-only sites, the freshest replica has the most up-to-date data (ideally as close to 1 as possible). The notion of freshness adopted in our approach will be discussed in detail in Chapter 7.

The third criterion is related to the concept of load and defines the "least loaded" site as the one with the smallest value of the load within a given set and uses the above defined notion of load.

The rationale behind choosing a best replica is the following situation that may require a replica selection: a particular heavyload situation requires the acquisition of a new update replica from its read-only children. For each replica s the replica selection algorithm keeps track of its load, $\mathtt{load}(s)$ and the freshness of each data object d, $\mathtt{fresh}(d, s)$. For read-only sites, the freshness is smaller or equal to one and represents a measure of the staleness of the data on a site. The algorithm begins by identifying for all replicas their load level, freshness level and the closeness to the requester and then chooses the best replica among them according to the following definition:

Definition 6.1 ***(Best replica function)*** *Let S be the set of all sites in the system and D the set of all data objects. Then, for a copy of a data object $d \in D$, required by a given client site $c \in S$, the function $\mathtt{best}(c, d)$ is defined as follows: $\mathtt{best} : S \times D \to S$, such that $\mathtt{best}(c, d) = s^* \in S$, where s^* corresponds to the greatest value of the sum: $\alpha\mathtt{close}(s^*, c) + \beta\mathtt{fresh}(d, s^*) + \gamma(1 - \mathtt{load}(s^*))$, for given α, β, γ with $\alpha + \beta + \gamma = 1, \forall s^* \in S$.* □

Choosing replicas in the round-robin manner would neglect the proximity factor. On the other hand, always choosing the closest replicas could result in poor load distribution. Our protocol keeps track of load bounds on replicas, freshness levels and proximity factors, which enables the replica management algorithm to make autonomous selection decisions. We allow the decision to take into account the freshness level of a replica. The usage of freshness in this protocol is a trade off of consistency for performance. Furthermore, by using a weighted sum for the notion of best replica, we allow higher level applications to give preference to a certain parameter over the others.

6.4 The Re:LOADiT Protocol

In the following we define the conditions and consequences of dynamic replica acquisition or release. When a new updateable replica is created, it is updated with the content of existing update replicas (depending on the data objects that exist at that site). In order to ensure consistency the creation of a new update replica takes place in two phases: (i) *Phase 1*: sites are informed that a new replica will join the network, they finish currently executing transactions and start queueing any direct operations belonging to subsequent transactions; (ii) *Phase 2*: the replica has joined and is up-to-date, the sites start executing the queued direct operations taking into account the new replica when executing indirect operations.

However, a new replica promote may not always be beneficial. The decision whether to acquire or release a replica is made based on a combination of local (accurate) and global (partially accurate) information, since we use a completely distributed approach. Nevertheless, in order to maintain consistency, the update replicas are synchronously informed of any replica promote or demote, which conforms to our eager replica maintenance approach for update site. The extensions required for the Re:LOADiT Protocol are schematically illustrated in Algorithms 6 and 7.

6.4.1 Replica Promote and Demote

In the following we explain how, when and based on which criteria replica management decisions are taken and the consequences that these decisions have on the state of the system. We first begin with a formal description of the replica promote and demote processes.

The process by which a read-only site is transformed into an update site is called **replica promote** (replica acquisition). The replica promote occurs under the following circumstances (formally called **Promote Precondition**): An update site s will begin the process of the acquisition of the best replica from its own read-only children, site r, which satisfies the criteria defined in Definition 6.1, if and only if the following statements hold: $\mathsf{load}(\mathsf{s}) = \mathsf{heavyload}$ and there is no other promote initiated by a different site taking place at the same time and the already existing number of update replicas in

Algorithm 6 Site Protocol (Extension)

```
 1: Main Execution Thread:
 2: while true do
 3:     Proceed normal execution;
 4:     wait for next message $m$;
 5:     if $m$ contains message: inform promote then
 6:         finish direct operations of active transactions;
 7:         update replica repository information;
 8:         return ACK message;
 9:         queue all incoming direct and indirect operations;
10:     else if $m$ contains message: end inform promote then
11:         execute queued operations;
12:     else if $m$ contains message: inform demote then
13:         update replica repository information;
14:     end if
15: end while
```

the system has not reached a maximum. In real-life scenarios it is not practical to allow the number of replicas to increase in an uncontrolled manner. We allow therefore the replication scheme to impose a maximum on the number of update replicas for a certain data object (nevertheless, if absolutely required, this maximum can be set to infinity). As a consequence the read-only site r has been promoted to an update site and all the other update sites are aware of the new replica. The information in the distributed replica repository is updated accordingly. The promote of a site can only then occur when all the other update sites have agreed. This constraint will ensure that multiple promotes do not occur at the same time and in an uncontrolled manner.

The process by which an update site is transformed into a read-only site is called **replica demote** (replica release). The replica demote occurs under the following circumstances (formally called **Promote Precondition**): An update site s will begin the process of self release for a data object x, if and only if the following statements hold: $load(s) = underload$ or load(s) = $overload$ and no active transactions exist at s and the data are available elsewhere at a minimum number of replicas. Here the same observation can be made: in real-life scenarios it is not practical to allow the number of replicas to decrease in an uncontrolled manner. We allow therefore the replication scheme to impose a minimum on the number of update replicas for a certain data object (nevertheless, if absolutely required, this maximum can be set to zero). As a consequence, site s no longer exists as update site and all the other update sites are made aware of the disappearance of the replica site s. The information in the distributed replica repository is updated accordingly. Site s would

Algorithm 7 Site Protocol (continued)

```
1: Background Thread:
2: while $\delta(T)$ do
3:    collect load
4:    if Promote Precondition then
5:       select best read-only replica according to Definition 6.1;
6:       inform update replicas of new promote;
7:       promote read-only replica;
8:       update replica repository information;
9:    else if Demote Precondition (in underload) then
10:      inform update replicas of self demote;
11:      update replica repository information;
12:   else if Demote Precondition (in overload) then
13:      if majority overload then
14:         inform update replicas of self demote;
15:         update replica repository information;
16:      else if minority overload then
17:         route read requests to sites $\notin overload$
18:      end if
19:   end if
20:   if (load - previous load) $> \Delta_{load}$ then
21:      $\delta(T)$ - -;
22:      propagate load changes to update replicas
23:   end if
24:   if (previous load - load) $> \Delta_{load}$ then
25:      $\delta(T)$ + +;
26:      propagate load changes to update replicas
27:   end if
28: end while
```

then become a read-only child of the best update replica among the update sites which were initially holding copies of the same data objects. Since an update site needs to be chosen as parent, the definition of best replica will reflect the notions of local load and the host proximity only (as all update sites have a freshness of 1). As in the previous case, the demote of a site can only then occur when all the other update sites have agreed. This constraint will ensure that multiple demotes do not occur at the same time in an uncontrolled manner.

Replica promote refers to the process according to which replicas move from the set of read-only sites to the set of update sites while replica demote refers to the process by which a site moves from the update sites to the read-only sites. These decisions are based on information locally stored on each site

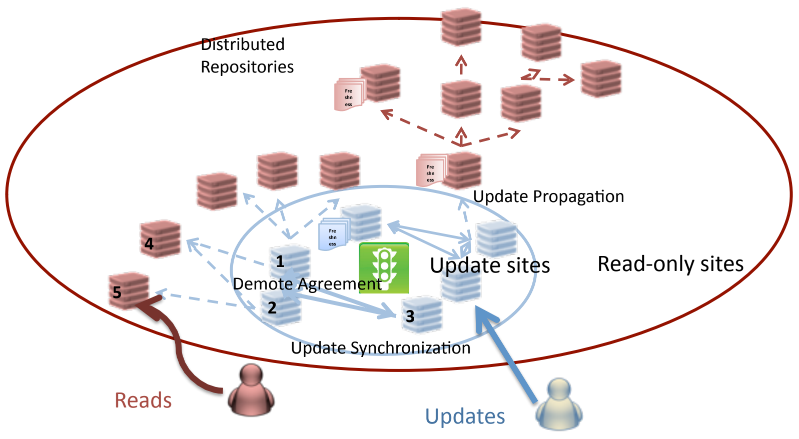


Figure 6.1: The Re:LOADiT Protocol - Demote Agreement in Underload.

in local repositories, information which is distributed between the sites, and replicated together with replica update synchronization, therefore no additional overhead is needed to replicate this information. The rationale behind why this information is bundled together with replica synchronization is that whenever updates are occurring there is information worth exchanging (the load on the update sites is more likely to vary, more updates that need to be propagated to read-only sites imply that freshness levels are also more likely to vary), so the repositories are updated as well. If no replica update is taking place, then there is no information worth exchanging anyway and, if at all needed, replica management decision can be taken using partly outdated data. Sites are consequently allowed to take decisions autonomously, based on their local information. Nevertheless the rest of the update sites need to agree unanimously, and all the sites obey the vote outcome. One important difference to be noted is that any site may be demoted to read-only site, but only the best replica for a certain situation will be promoted to update sites.

As already mentioned, the local load of a site heavily influences our Re:LOADiT protocol. The replica promote and demote decisions are taken based on load trends that fall into a certain load interval. In the following we give a detailed description of the load intervals and the replica decisions made according to each load interval by using again the example introduced in Chapter 2. As our protocol is completely decentralized, replica decisions are made mostly based on local information.

Let's assume a network structure as introduced in Chapter 4 with update and read-only sites and different types of users. Update operations belong to user update transactions and will be directed by the middleware to update sites. Read operations may belong to either user update transactions or user queries and may consequently be directed to either update or read-only sites.

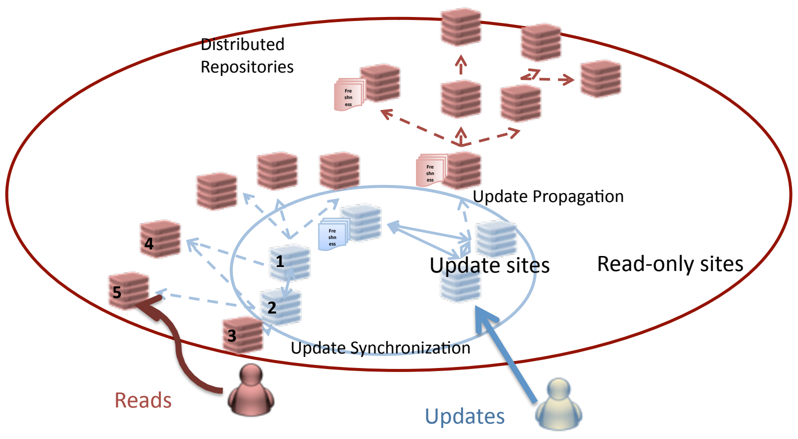


Figure 6.2: The Re:LOADiT Protocol - System State after Demote.

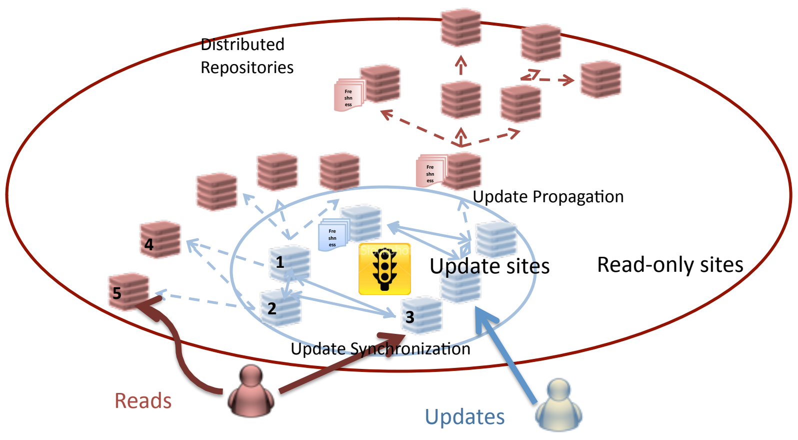


Figure 6.3: The Re:LOADiT Protocol - System State in Lowload.

An **underload** situation intuitively implies that there are no updates and no read operations at this site. This case would occur for example in case a scientist is monitoring a region where incidents are not likely to occur. Consequently, other scientists are less likely to be interested in the same region, and user requests for these data are seldom. This case is graphically presented in Figure 6.1. In this situation for example site 3 autonomously decides to self demote (as long as the replication scheme allows it and the preconditions for demote are fulfilled). It sends a demote agreement request to the other update sites which hold copies of the same data objects (in this example site 1 and 2). This information is available in the replicated replica catalog. Once the other update sites have agreed on the demote, site 3 demotes itself to read-only and will no longer be part of the replica synchronization process. This case is illustrated in Figure 6.2.

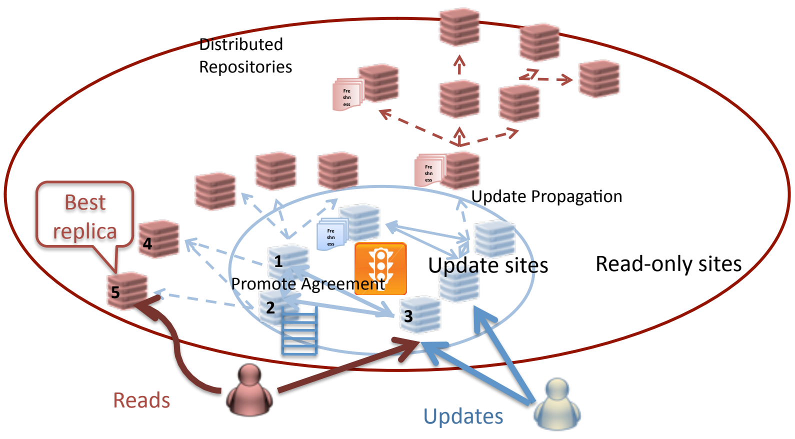


Figure 6.4: The Re:LOADiT Protocol - Promote Agreement in Heavyload.

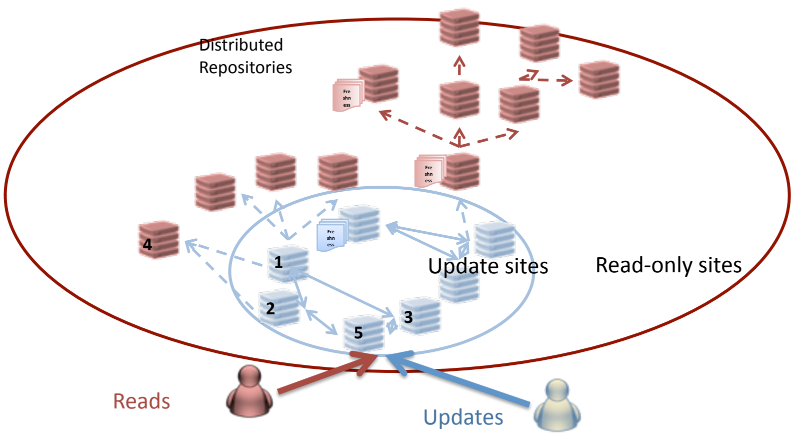


Figure 6.5: The Re:LOADiT Protocol - System State after Promote.

A **lowload** situation is considered to be a normal load situation. The sites continue their normal execution. As this situation is considered to be the normal functioning load of a site, the replication protocol will maintain the status quo. This case is graphically illustrated in Figure 6.3.

Assume now that due to extra load site 3 is in a **heavyload** situation as presented in Figure 6.4. The replication protocol will dictate that it requires the promote of a read-only site to share its load. Site 3 will check among his children to find the read-only replica site with the freshest replica, the closest geographically or the least loaded, or a combination of the three, in other words the best replica (for this situation). Assume that the best replica is in this case read-only site 5. In the first phase update sites 1 and 2 are informed that a new replica will join the network. If they agree, they will finish currently executing transactions and start queuing any direct operations belonging to subsequent transactions in their local queues (Figure 6.4). In the

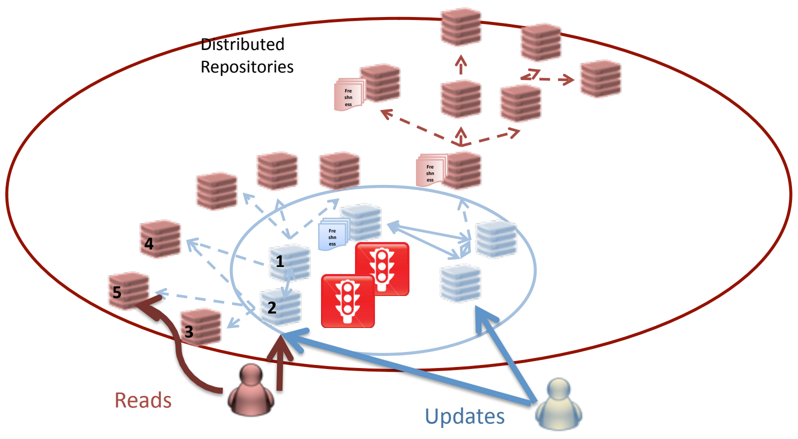


Figure 6.6: The Re:LOADiT Protocol - Replica Demote in Overload.

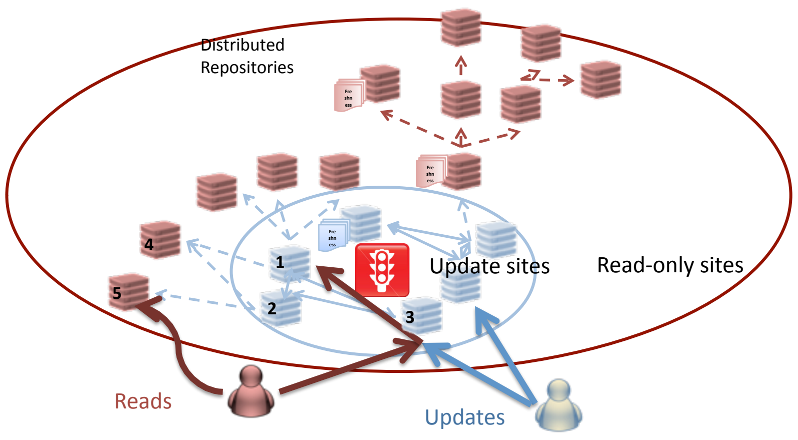


Figure 6.7: The Re:LOADiT Protocol - Request Routing in Overload.

second phase, depicted in Figure 6.5 site 5 has joined the update sites and is up-to-date. The sites start executing the queued direct operations taking into account site 5 when executing indirect operations.

When the high load level is crossed the sites enter into what we call **overload** situations, in which their continuing to function under these conditions might affect performance. An overload situation is a special case in which the global state of the system needs to be taken into account, since a local situation may be the result of several influencing factors that do not necessarily reflect the state of the system. We distinguish two sub-cases: In case the system is in a majority overload situation, the acquisition of a new replica is unlikely to improve the situation (since more replicas imply more synchronization). In this case, a replica demote might prove to be beneficial in the long run, as shown in Figure 6.6. In case the system is in a minority overload situation, the overload can be assumed to be due to read operations (which are not replicated). The protocol will dictate the site(s) in overload to migrate the read request to other update sites in the system which are not in an overload situation, as presented in Figure 6.7. The reasons for this behavior are the following: updates contribute to the global load, therefore in this case we need to take into account global load levels as well. Reads on the other hand only increase the local load on the site.

It is important to notice the following aspect of our protocol: The Re:SYNCiT protocol produces a globally serializable schedule (although no central scheduler exists and thus no schedule is materialized in the system) [183, 187, 186]. By using a two-phased approach when replica promote and demote decisions are made, the serializability is guaranteed. In Re:LOADiT (which builds further functionality on top of Re:SYNCiY) data consistency in ensured, even in the presence of dynamic replication.

6.4.2 Failure Handling

We identify the following types of failure that could occur in dynamic data Grid environments. We suppose the sites fail-stop and that failures are detected by means of failure to acknowledge requests within a certain timeout. This timeout mechanism ensures that failure detection prevents processes from long delays but also, from suspecting sites too early which could incur unnecessary communication overhead. If a site crashes during an inform demote or inform promote phase, the failure is detected by means of the timeout mechanism. The site in charge of the inform will update the replica repository and reissue the inform request. If the site to be chosen promoted is failing to answer, the next best replica (according to Definition 6.1) is chosen to be promoted. Since the number of copies of the object varies in time the dynamic replication and allocation algorithm is vulnerable to failures that may render the object inaccessible. To address this problem, we impose a reliability constraints of the following form: “The number of copies per data object cannot decrease below a minimum threshold". If such constraint is present, and the local site s fails to meet it, then any of sites informed of the demote p refuses to accept the exit of a data site, if such exit will downsize the replication scheme below the threshold. In other words, p informs s that the request to exit from the replication scheme is denied; subsequently, updates continue to be propagated to s. Site s continues to reissue the request whenever the preconditions for replica demote dictates to do so. The request may be granted later on, if the replication scheme is expanded in the meantime.

6.5 Comparison to Existing Approaches

A replicated database built for high availability must eliminate all single points of failure. Load balancing is typically intrinsically tied to the replication middleware. Load balancing can be implemented at the connection level, transaction level or query level. Connection-level load balancing allocates new client connections to replicas according to a specified policy; all transactions and requests on that connection go to the same replica until the connection is closed. This approach is simple, but offers poor balancing when clients use connection pools or persistent connections. Transaction-level or query-level load balancing perform finer grain load balancing by directing queries on a transaction or query basis, respectively. As an example, Tashkent+ [72] provides transaction-level load balancing and exploits knowledge of the working sets of transactions to allow in-main-memory execution at every replica. The result is an improved throughput of more than 50% over previous techniques; however, the approach uses a centralized load balancer that is not replicated. A failure of this component could collapse the entire system.

SNOWBALL [36, 180], is an example of load balanced data management algorithm which supports distributed storage and retrieval of keyed, record-structured data on networks of workstations. It consists of an approach to distributed data access structures, which is completely decentralized, provides scalable cost/performance, and copes with evolving access patterns, but so far only considering closeness as a criterion for replica selection and placement.

Dynamic replication in distributed environments has known extensive development over the years. Approaches such as the ones presented in [11, 56, 88, 153] have been successfully applied to peer-to-peer networks. Sophisticated solutions, which eliminate the need of a central coordinator, have been proposed, but they either do not take into account data freshness [11], or multiple data types ([153] for example, only considers images as data objects) or neglect consistency issues [88]. In most cases results are only applicable to a certain type of network: hierarchical namespaces or unstructured P2P systems, or neglect consistency issues [88].

Early attempts to evaluate the scalability properties of adaptive algorithms for replica placement services in dynamic, distributed systems were reported in [42, 182, 15]. Weissman and Lee [193] proposed a replica management system which dynamically allocates replicated resources on the basis of user demand, where resource requests by users can be transparently rerouted if individual replicas fail. Testbed experiments demonstrated the scalability of this approach. The SRIRAM research system [179] for automatic replication of computing resources in Grids and other distributed environments was designed to improve resource availability and fault tolerance. Here, computing resources were members of networks, or meshes, which could be searched to find nodes on which Grid processes could be replicated. The search of large meshes was made more efficient through the organization of participant resources in a spanning tree structure and through intermediate caching of query results for reuse. The spanning tree is automatically reconfigured as nodes are added or removed, allowing the system to scale and respond to dynamic conditions. Participant resources operated securely and anonymously, allowing the mesh to incorporate multiple administrative domains. More recently, other replication schemes have been proposed. Through experiments, most have demonstrated limited scalability and ability to operate under conditions of resource heterogeneity and dynamism. Valcarenghi [178] presented a replication approach in which replicas are located in proximity to each other to form service islands in a Grid network. Different replica configurations were evaluated using a Mixed Integer Linear Programming model to determine which configuration of islands exhibits higher fault tolerance. Simulations demonstrated that the approach can enable recovery of a high percentage of long distance inter-service connections, while minimizing the number of replicas needed and thus simplifying replica management. These approaches have been developed at the service level and thus do not take into account data replication constraints or freshness and versioning issues.

In [115], a resource allocation system for a computing Grid used in a telecommunications company was reported. In this system, dynamic process replication was used to provide fault tolerance and enable fulfillment of terms of service level agreements. Within the e-Demand project, [198] proposed a replication method that detected faulty computations in Grid workflows consisting of multiple tasks. Here, a workflow was simultaneously executed by different sets of service replicas. A voting process was used to select which replica set should return its result to the user. This approach also facilitated identification of faulty services that failed in more than one workflow, allowing the service to be eliminated from future consideration. Testbed experiments demonstrated that this approach improved workflow fault tolerance. Genaud and Rattanapoka [85] developed a mechanism for MPI-based Grid environments that used resource replication to increase fault-tolerance of parallel computations and demonstrated limited scalability in experiments. Other methods for replicating computations on resources have been proposed in [2, 39] and have been tested experimentally. The test results show that they are not scalable for large data Grids.

Less work appears to have been done of efficient and scalable methods for synchronization of replica states. One method, based on selective replica placement, proposed by [178], is described above. In [199], this issue was investigated for service replicas that exhibit non-deterministic behavior and use asynchronous messaging. Here, the researchers proposed an optimized version of the Paxos algorithm [116] for synchronizing replicas in distributed environments and demonstrated the efficiency of their approach under both local and wide-area conditions. In earlier work [200], a more traditional primary-backup approach was used to investigate replication of Grid services that were implemented using Open Grid Services Infrastructure (OGSI) [176] and the Globus Toolkit [98]. In [200], it was found that the strategy could be readily implemented and resulted in higher service availability in local area environments. However, the overhead costs imposed by OGSI notification in order to synchronize states of service replicas that behaved non-deterministically were significant. The study showed that the overhead associated with replica synchronization can be eliminated by allowing failed tasks to be restarted on replicated resources reserved for this purpose. To date, this work has not been repeated with successor specifications to OGSI. Dasgupta et al. [64] proposed a framework for incorporating reservation of redundant backup resources into service-level agreements where failure of the primary allows switching to a backup. Simulation showed circumstances where this approach improved efficiency of system resource allocation. Finding efficient and scalable methods for replica synchronization remains a challenge that must be met before resource replication can be fully utilized as a fault-tolerance tool in Grid environments.

6.6 Summary

The nearly unlimited storage capabilities of Data Grids allow data to be replicated at different sites in order to guarantee a high degree of availability. For updateable data objects, several replicas per object need to be maintained in an eager way. The number of updateable replicas has to be dynamically adapted to optimize the trade-off between synchronization overhead and the gain which can be achieved by balancing the load of update transactions. Due to the particular characteristics of the Grid, especially due to the absence of a global coordinator, replication management needs to be provided in a completely distributed way. This includes the synchronization of concurrent updates as well as the dynamic deployment and undeployment of replicas based on actual access characteristics which might change over time.

In this chapter we have introduced the Re:LOADiT approach to dynamic replica deployment and undeployment in the Grid. Based on a combination of local load statistics, proximity and data access patterns, Re:LOADiT dynamically adds new replicas or removes existing ones without impacting global correctness.

7

Freshness Requirements for Data Grid Replication

In the previous chapters we have presented a new approach to dynamic replication in a data Grid with provably correct transactional guarantees. This approach dictates how update site behave and from a user's point of view the clients will always access the most up-to-date data. This chapter will introduce the Re:FRESHiT protocol, which allows to effectively trade freshness for performance and addresses freshness and versioning issues, needed in many Grid application domains, without losing consistency. Re:FRESHiT is based on the notion of freshness, which indicates how much a read-only site has deviated compared to the up-to-date version present on the update site. We use, as before, eager replication mechanisms to ensure replica consistency for update sites. Updates are propagated in a lazy manner to the read-only sites by means of decoupled refresh transactions. Propagation transactions are continuously scheduled during a site's idle time to reduce the work of refresh transactions whenever needed. Queries with different freshness levels are cleverly routed along our site topology, by taking advantage of its tree structure.

7.1 Problem Statement

Novel applications in eScience previously introduced in Chapter 2 have stressed the need to take into account particular requirements: (i) to manage and store large volumes of data, (ii.) to update (parts) of the data as new findings become available, (iii.) to provide this functionality in a completely distributed way, (iv.) to provide access to the most recent version of all data items, and (v.) to also keep outdated versions for read access. The first issues have already been solved by the protocols detailed in Chapters 5 and 6. For the latter, the notion of data freshness needs to be supported in order to allow users to specify the staleness of data, i.e., how old the data they require can be. Depending on a user's freshness requirements, for example, "at most twenty minutes' old data", finding the best replica does not need to suffer from the replica synchronization overhead, but still can provide consistent access to data. Coming back to the scenario introduced in Chapter 2, consider the case of several scientists at different research institutes, performing oil slick distribution studies, in order to determine the environmental impacts of stranded oil, and offer recommendations of cleanup procedures and methods least likely to exacerbate the effects caused by the oil spil. In order to perform a thorough investigation they require several successive data acquisitions (before and after the oil spill), combined with wind, sea-state and other meteorological data for complementing the information sources. Older versions, from multiple archives of satellite images, and previous works of other scientists, are an acceptable solution which guarantee faster access to older, but still consistent data. Over longer time scales, this type of damage assessments provides basis for monitoring and recovery assessment programs, and is of interest to scientists all over the world.

7.2 Freshness Metrics

Freshness measures are closely related to the notion of coherency for which several ideas have been proposed in the literature [54, 137, 157]. We use the notions of *absolute freshness* and *delay freshness* to characterize our freshness function and apply it to our network structure.

Definition 7.1 ***(Absolute Freshness)*** *The absolute freshness of a data object* d *is defined by means of the time* $\tau(\mathrm{d})$ *of the last committed update transaction that has updated* d. □

The use of absolute freshness implies that the younger the timestamp, the fresher the data. Timestamps are values which are strongly monotonically increasing. This implies that a data object is newer the more recently it has been created. As time and time stamps are (strongly monotonically) increasing, this means the larger $\tau(\mathrm{d})$, the fresher (younger) the data.

In a replicated system with several replicas and a lazy replication approach, each replica might have a data object (replica) with a different absolute freshness - the freshest being the one where the last update transaction has committed.

Definition 7.2 ***(Delay Freshness)*** *A delay freshness defines how late in time a certain read-only site is compared to an update site that holds a copy of the same data object. We define* $\tau(d)$ *to be the commit time of the last refresh transaction that updates a copy of a data object* d *on a read-only site, and* $\tau(d_0)$ *the commit time of the most recent update transaction on the update site that updates* d*. Then the freshness function is defined as* $f(d) = \frac{\tau(d)}{\tau(d_0)}$*, with* $f(d) \in [0, 1]$. □

The delay function reflects how much the data has deviated from the up-to-date version. Intuitively, a freshness of 1 means the data are up-to-date, while an index of 0 represents "infinitely" outdated data, i.e. the data are not present on the site.

We assume that each update (Level 1) site assigns to every committed transactions a unique timestamp which is greater than all timestamps assigned to previously committed transactions. Note that since we assume some form of clock synchronization between the update sites, clock skew can be ignored. Together with load information [185], each site propagates downwards in the tree structure the freshness timestamps (of the last propagated update from the update site), such that at each level sites are aware of their predecessors and/or successors load and freshness information.

A read-only site s is said to hold a copy of a data object d which fulfills a freshness level required by a client c, if $\tau_s(d) < \tau_c(d)$. Refresh transactions are used to bring all sites which hold data objects required by the client transaction to the same freshness level by executing a sequence of update operations. Running a refresh transaction has the same effect as sequentially running the remaining propagation transactions at the site as long they the follow the tree structure.

The freshness function is monotonically decreasing in the tree, from the root to its leaves. The further down the path in the tree structure we go, the less accurate the freshness function becomes. This is an important observation upon which our routing strategy is based. Read-only sites at all levels are able to determine using their own local knowledge whether the current version of the data can be used to serve a user request of the type: *"I am fine with yesterday's data / data as of 12 o'clock last Monday"* or if this request needs to be routed to a predecessor or a successor in the tree hierarchy. In other words, all read-only sites are able to autonomously decide whether they satisfy a required freshness level and where to route the request if not.

7.3 The Re:FRESHiT Protocol

Re:FRESHiT uses decoupled refresh transactions to propagate updates through the system, on-demand, in order to bring the read-only replicas to the freshness level specified by the read-only transactions [10, 157]. Propagation transactions are performed during the idle time of a site in order to propagate updates to the read-only replicas. Therefore, propagation transactions are continuously running as long as there is no running read or refresh transaction. Re:FRESHiT exploits the read-only sites' idle time by continuously scheduling propagation transactions as update transactions at the update sites commit. This way, copies at the read-only sites are kept as up-to-date as possible, such that the work of refresh transactions (whenever needed) is reduced and the overall performance is increased.

The Re:SYNCiT protocol produces a globally serializable schedule (although no central scheduler exists and thus no schedule is materialized in the system) [186]. Moreover, the update transactions' serialization order is their commit order. Each propagation transaction inherits the timestamp of the committed update transaction and this timestamp is propagates to all read-only sites. The extension required by the Re:FRESHiT protocol in order to support freshness requirements without losing consistency are summarized in Algorithm 8.

We use *freshness locks* [10, 157] in order to prevent propagation and/or refresh transactions to update a data object above the freshness level required by an on-going read-only transaction. Freshness locks are placed on the objects at the read-only sites in order to ensure that replica maintenance does not overwrite the versions needed by a running transaction. A freshness lock placed on a data object d at a site s with an associated absolute freshness, required by a running read-only transaction, will not allow an update operation on the data object d at site s to bring this data object to a younger absolute freshness. Similar to the approach in [10], we define freshness locks to be compatible in the sense that different read-only transactions may acquire freshness locks on the same data object, possibly with different freshness levels, as long as the site's data are fresh enough to serve the requests.

We make no assumption regarding the scheduling of read-only transactions, that is, we allow users to directly query any read-only site in the network. Furthermore, we allow users to specify freshness requirements as quality of service constraints. However, these requirements might be implicitly changed by the middleware if none of the objects involved in the transaction satisfies the required freshness level.

We foresee the following situations which may occur during the lifetime of a read-only site, s:

Algorithm 8 Site Protocol (Extension)

```
1:  Scheduling Thread:
2:  while true do
3:     //receive a user request for a data object $d$ with a certain freshness level
4:     if $d$ is not replicated locally then
5:        // find a site $s_{out}$ which contains a copy of $d$ from the replica repository
6:        route request to $s_{out}$
7:     else if $f_{site}(d) \geq f_{client}(d)$ and site $s$ is not in overload then
8:        execute read locally
9:     else if $f_{site}(d) \geq f_{client}(d)$ and site $s$ is in overload then
10:       //check if at least a child is the best replica;
11:       if child = best(c,d) then
12:          route request to child not in overload
13:       else if no site to route then
14:          execute read locally;
15:       end if
16:    else if $f_{site}(d) < f_{client}(d)$ then
17:       //check if at least a (transitive) parent is the best replica;
18:       if parent = best(c,d) then
19:          route request to parent;
20:          recursive check going up the path
21:       else if no site to route then
22:          request refresh transaction;
23:       end if
24:    end if
25: end while
```

1. *Read requests that can be processed locally:* A read-only transaction T_j submits a read operation $r_j(d)$ to a read-only site s that stores a copy of d. The following rules apply:

   - If $f_s(d) \geq f_{T_j}(d)$ and s is not in an overload situation, then the read operation is executed. If $f_s(d) \geq f_{T_j}(d)$ and s is in an overload situation, then the read operation is re-routed downward in the tree until the best site s_l is found which fulfills the condition $f_{s_l}(d) \geq f_{T_j}(d)$ and s_l is not in an overload situation. In our work, we use the definition of best replica previously introduced in [185], giving preference to load and freshness metrics. Accordingly, the best site fulfills the required freshness level and is not in an overload situation. Very importantly the downward propagation in the tree is possible as each site in the tree is aware the freshness of its children.

2. *Read requests that cannot be processed locally:* A read-only transaction T_j submits a read operation $r_j(d)$ to a read-only site s that stores a copy of d. The following rules apply:

   - If $f_s(d) < f_{T_j}(d)$, then the read operation is re-routed upward in the tree until the best site s_m is found among the (transitive) parents of s which fulfills the condition $f_{s_m}(d) \geq f_{T_j}(d)$ and s_m is not in an overload situation. If no such site is found, s requests a refresh transaction until $f_s(d) \geq f_{T_j}(d)$. The read operation can be processed on s after the refresh transaction has been executed on s. We thus take advantage of the tree structure and are able to route requests upward or downward in the tree to the best replica able to process the request. Since the freshness is monotonically decreasing in each tree from the root to the leaves, each site is able to route a request appropriately, either up or down to the site that is best able to process the request. In order to preserve the freshness monotony, when s requests a refresh transaction it becomes a Level 2 direct child of its root update parent. The dynamic restructuring of the tree is detailed in Subsection 7.3.1.

3. *Read requests for data that are not replicated locally:* A read-only transaction T_j submits a read operation $r_j(d)$ to a read-only site s that does not store a copy of d. Then the read operation is re-routed to a site in the tree that contains a copy of that data object, from the replica repository. The replicated replica repository information is passed along through the tree structure together with propagation transactions from the Level 1 update sites.

4. *No read requests:* In this case only propagation transactions can occur at s. Propagation transactions execute changes from the local propagation queues (where updates propagated downwards from the update sits are stored). S delays a propagation transaction P until all propagation transactions with smaller freshness levels than P have committed at s. A propagation transaction is dropped if the site has already seen and processed a propagation transaction with the same or higher freshness level or a refresh transaction.

We demonstrate the Re:FRESHiT protocol by using the following example. We again come back to the scenario introduced in Section 2, in which Scientist 2 requires several data acquisitions (before and after the

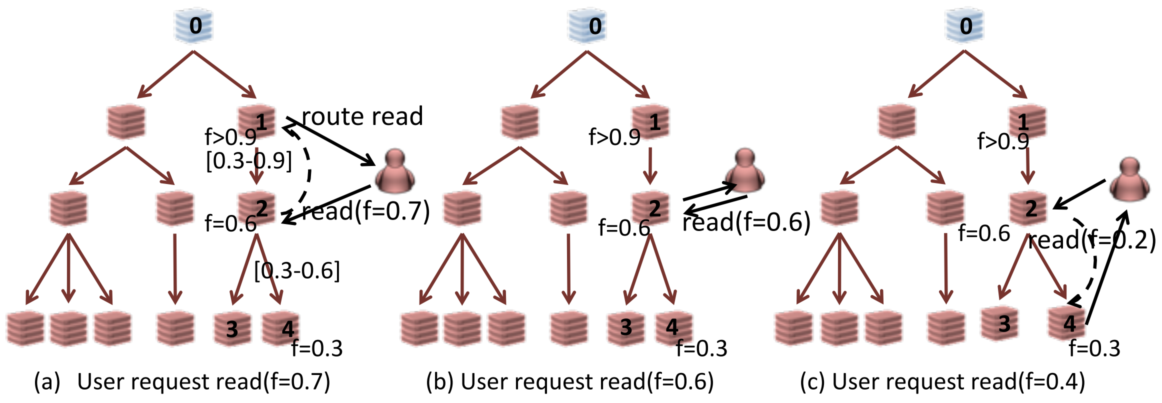

0
1
2
3
4
route read
f>0.9
[0.3-0.9]
f=0.6
read(f=0.7)
[0.3-0.6]
f=0.3
(a) User request read(f=0.7)
read(f=0.6)
(b) User request read(f=0.6)
read(f=0.2)
(c) User request read(f=0.4)

Figure 7.1: Propagation and Refresh Transactions Example for the Read-only Sites.

oil spill). This can be schematically represented as the following query: $T_1 : read(d_1, "yesterday") \quad read(d_2, "yesterday")$.

Assume that this query is made at the Level 3 read-only site 2 in Figure 7.1, which contains outdated data. Assume furthermore that site 2 is not able to serve the request for data object d_1 with freshness f_1 (as of yesterday). According to Algorithm 8, site 2 will check its parent site to see if the required freshness criterion is met. If site 1 is the best replica, that is, it satisfies the required freshness and its load level (as defined in 6.2) allows it, then the request will be routed to site 1 (see Figure 7.1 (a)). If site 2 is able to service the request it will do so, as shown in Figure 7.1 (b). Assume that another user would request an even staler data than stored at site 2 (case illustrated in Figure 7.1 (c)). Since site 2 is aware of the freshness levels of its children and since obviously (due to the tree structure) the children sites have staler data, site 2 will re-route the request to site 4, which can process the request locally. If no such site exists among site 2's children the request would be processed locally, and the user receives fresher data than requested.

Assume a request arrives at a site with an even higher freshness level. The request is propagated upwards until a suitable site would be found. If no site is found, then according to the algorithm site 2 would request a refresh transaction in order to be updated with the most up-to-date data from the root update site (site 0). Therefore users are not guaranteed to obtain the exactly specified freshness level (as of yesterday), but they will receive data that are at least as old as the specified freshness level (in this case, today's data). The following subsection describes how this situation is handled and the changes required in order to preserve the monotony of the freshness in the tree.

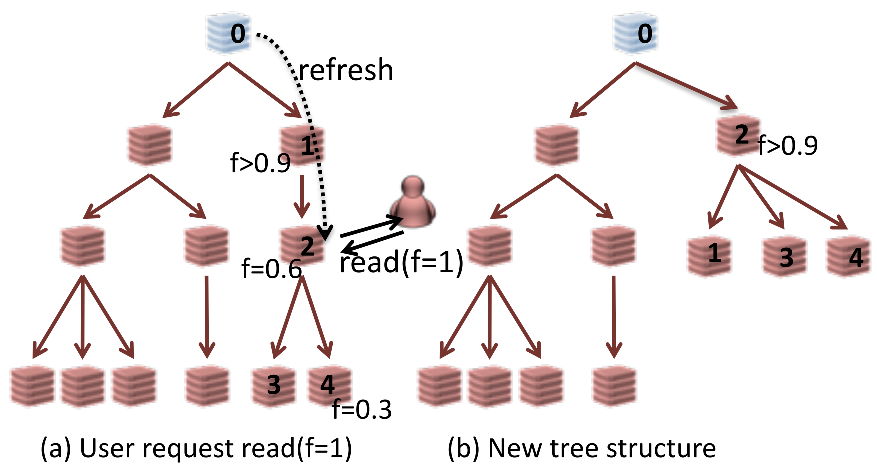


Figure 7.2: Dynamic Change of the Tree Structure.

7.3.1 Tree Dynamics

The tree structure that we assume in our approach is not static, and can change dynamically. Assume a query with a higher freshness level (than the site can service using its local data) arrives at a Level 3 node. If none of the sites in the tree path is capable of servicing this request, the Level 3 site will request a refresh transaction from its top most Level 1 parent. It will consequently become the read-only site with the freshest data in the path. Leaving it in the current place in the tree hierarchy would no longer guarantee the monotony of the freshness. This Level 3 site will consequently become a direct (Level 2) child of its root (Level 1) update parent.

Let's consider the following example. We come back again to the use case scenario from the earth observation domain in Section 2, in which Scientist 2 requires several data acquisitions (before and after the oil spill). For his data acquisition after the oil spill he requires the most up-to-date data. Assume that this query is sent to the Level 3 read-only site 2 as illustrated in Figure 7.2, which contains outdated data. Assume, furthermore, that site 2 is not able to serve the request for the most up-to-date data. Site 2 will check among its predecessors in the tree hierarchy to see if there is a site capable of serving this request. If no site is found (case described in Figure 7.2 (a)), then site 2 would request a refresh transaction and it will be updated with the most up-to-date data from the root update site (site 0). Consequently, it follows in a logical manner that site 2 will become a direct child of site 0 and itself a Level 2 site, thus rotating the tree structure in order to preserve the monotony of the freshness in the tree (see Figure 7.2 (b)).

It now becomes obvious that propagation transactions always follow the paths in the tree structure. Refresh transactions on the other hand can change (but not necessarily) the tree topology.

7.3.2 Failure Handling

In order to prevent failure situations, we use redundant links between sites in different levels (see Figure 4.5). Nevertheless, propagation transactions and query re-routing always follow the tree structure.

If a new site s wants to join as read-only site for a data object d, it selects a site in a tree for d according to rule 3 given in the Re:FRESHiT Algorithm described in Section 7.3. Let s_k be the selected site. Then, s's join request is forwarded downwards in the tree, starting from s_k, by always selecting the freshest subordinate until a leaf site s_l is reached. Then, s is added as child site to s_l and it is updated with s_l's copy. In a similar way, new sites are recruited if the number of replicas needs to be increased for a certain data object.

7.4 Comparison to Existing Approaches

According to Brewer's CAP theorem [37], Strong *Consistency*, High *Availability* and *Partition Tolerance* are three requirements that exist in a special relationship when it comes to designing and deploying applications in a distributed environment as only two of the three properties can be fulfilled at the same time. We relax therefore the notion on consistency in our approach and allow users to read data with different freshness levels, as required in the example application scenario presented in Section 2.

Different approaches to replication management have been studied so far. Eager replication is a standard approach used in the database community, which typically uses two-phase commit in combination with two-phase locking to guarantee one-copy serializability and replica coherency [28, 110]. It has been argued that this approach provides unacceptable performance as soon as the update rate or the number of copies increases [92]. The main drawback of eager replication management is that all copies of a data item are maintained within the same transaction which hinders scalability and prevents from efficient executions. Therefore, eager replication is not a viable approach in most of current data processing environments, especially when being applied at a Grid-scale.

Previous work on lazy replication management decouples replica maintenance from the original transaction [33, 55, 136]. In other words, transactions keeping replica up-to-date and consistent run as separate and independent transactions after the original transaction has committed. Although additional efforts are necessary to guarantee serializable executions, approaches presented in [33, 32] suggest a suite of lazy replication protocols that have lead to significant performance improvements and guaranteeing one-copy serializability. Previous work on lazy replication has concentrated on performance and correctness only. As observed from the example application scenario, today's applications have different requirements. [157] addresses some of these issues with the additional notion of freshness. It allows read-only clients to define freshness requirements stating how up-to-date data items must be when being accessed. However, the approach requires a centralized coordination component for scheduling and bookkeeping which is a potential bottleneck and a single point of failure. Second, it has only considered full replication at a granularity of complete databases. Clearly, this is way too inflexible as it precludes more sophisticated physical data organization schemes such as partial replication, partitioning or striping across sites. Third, previous work on lazy replication like [32, 134, 138, 170] assumed that the transaction executes entirely at its initiation site, which may not be the case in practical settings.

A recent protocol for a finer grained data replication that supports freshness and lazy update propagation for many read-only nodes has been presented in [10] . In [9], an adaptation of this protocol to the Grid is presented. However, this protocol suffers from a strict assumption on the existence of a central component which is used to collect and serialize all updates at the update nodes. Whenever a read-only node requires fresh data, it goes to this central component to get what it needs. This is conceptually equal to having only one update node in the system, which is a potential bottleneck and a single point of failure, and therefore is not practical in a Grid environment. In [14], several updateable replicas are supported but a single, global replication graph is required. This graph corresponds to a global agreement of all nodes holding replicas on the order of propagating updates in the system. Thus, none of the existing replication protocols can be fully decentralized which, in turn, is an important requirement in infrastructures like the Grid.

7.5 Summary

Data Grids are becoming more and more popular since, in contrast to traditional approaches, their nearly unlimited storage capacity allows tackling the data management problems in eScience. In particular, data Grids allow data to be replicated across different sites in order to increase availability and, by proper replica placement, to bring data closer to their users. However, the degree of replication is constrained by concurrent updates as synchronous replication with a large number of copies does not scale. This necessitates a separation between updateable replicas (updated synchronously) and many read-only copies which might hold stale data, characterized by a freshness level that indicates the deviation from the most recent version. Allowing users to specify freshness requirements for their queries significantly facilitates the routing of queries to read-only sites in the data Grid. In this chapter we have presented Re:FRESHiT, a freshness-aware replica selection and maintenance protocol for a data Grid based on a distributed dynamic replica management approach to updateable replicas without any global coordinator. Starting with a core group of updateable copies, read only copies with different levels of freshness are organized in a hierarchical structure. Re:FRESHiT supports the freshness-aware routing of queries in the Grid and also takes into account the sites' local load for replica selection without relying on any central component. In parallel, updates are propagated from updateable copies to read-only copies along the site hierarchy in a consistent manner.

8

Prototype Implementation and Evaluation Setup

The following two chapters present the results of the experimental evaluation of the Re:GRIDiT family of protocols presented in this thesis and the implementation details of Re:GRIDiT. The Re:GRIDiT protocols have been implemented using state-of-the-art Web service technologies which allows an easy and seamless deployment in any Grid environment. The evaluation has been conducted on up to 48 update sites and 48 read-only site. We have used simulated workloads that mimic the behavior expected from our use case applications. We begin by giving a brief introduction into the framework used for implementing the protocols and some implementation details, justifying the implementation choices that we made. We then provide an overview of the evaluation setup and the performance metrics used for the evaluation. Chapter 9 will detail the discussion of the evaluation results.

8.1 The Globus Toolkit Development Framework

The Globus Toolkit's Java Web Services Core (Java WS Core) is a Java development kit for building stateful Web services based on the WS-Resource framework. It also provides a lightweight hosting environment for such services. Java WS Core manages the life cycle of services and their resources, provides persistence support, and offers advanced security features. It also provides facilities for starting periodic and background tasks and has a unified way to store and retrieve service configuration data. At runtime Java WS Core provides an internal JNDI-based registry with service-specific and other configuration information. Web services can use this registry to look up configuration information, communicate with other services, or discover container-provided facilities. The security features of Java WS Core include a pluggable authorization engine, declarative and programmatic security, and multiple authentication mechanisms. The programming model of Java WS Core is flexible, letting the service developer choose small service-building blocks and combine them to create a service with the exact functionality the developer wants to provide. It also allows the developer to customize the built-in functionality or substitute it with an alternative implementation. The programming model of Java WS Core decouples the Web service (business logic) functionality from the resource (state). The web service implementation is usually a plain, stateless Java object. A service can be composed from several independent Web service operation implementations, called "operation providers". These operation providers enable easy reuse of common Web service operations among different services. We chose Java WS Core because it is created with standard Apache software components such as Apache Axis 1 (SOAP processor), Apache Addressing (WS-Addressing support), and Apache WSS4J (message security support) and is an open source project licensed under an open license. In order to implement stateful Web services, the WSRF (WS-Resource Framework) standard [78] has been proposed and adopted by the Globus Toolkit Framework. For performance, as well as bookkeeping reasons, we have implemented the state of the main web service components using Apache Derby database tables [67]. Other logging information, used for recovery and failure handling purposes, is also stored in the local databases. The choice of Derby is motivated by the following advantages: it is a lightweight, pure-Java relational database that can be embedded directly into Java applications and/or accessed remotely by multiple users connected to Derby's network server. However, any other existing off-the-shelf database can be used for this purpose.

8.2 The Globus Toolkit Java Core Services

The Globus Toolkit version 4 (GT4) Java core services offer a run-time environment capable of hosting Grid services. The run-time environment mediates between the user-defined application services and the GT4 core services, underlying network, and transport protocol engines. GT4 also provides development support, including programming models for exposing and accessing Grid service implementations. One of the compelling reasons to use GT4 is that it builds upon existing web services standards and technologies like SOAP and WSDL. All of the Grid service interfaces are exposed in WSDL

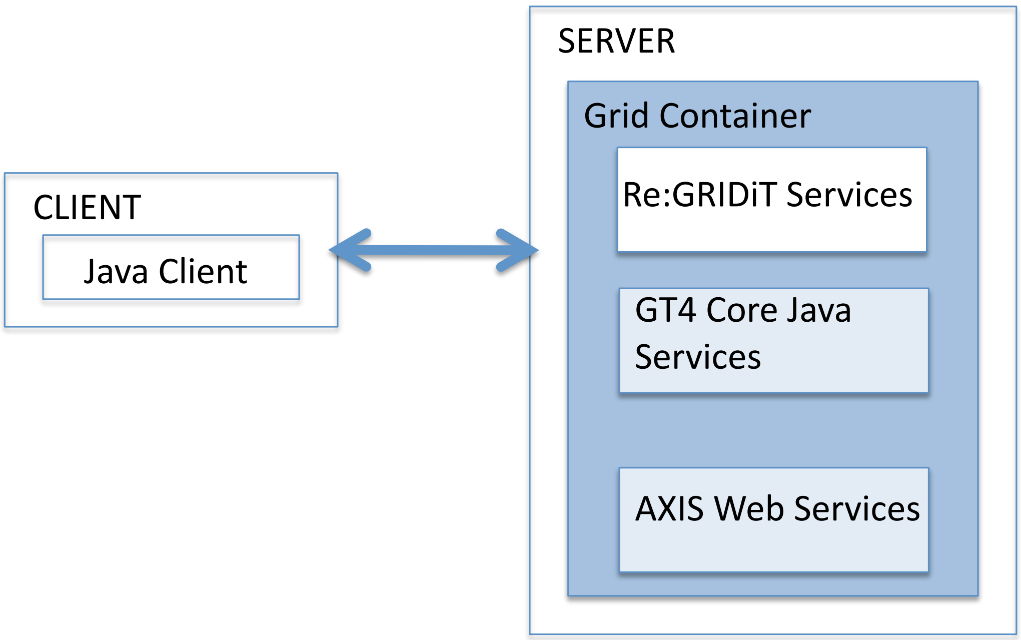


Figure 8.1: GT4 Java architecture

format. GT4 provides software libraries that support security, discovery, resource management, invocation, communication, exception handling, etc.

Figure 8.1 shows the major architectural components of the server side of GT4. This is just a subset of the functionality that GT4 provides and that we used in the implementation. The GT4 architecture consists of a Grid container to manage all of the deployed web services throughout their life cycles. The GT4 Grid container uses Apache AXIS as its web services engine to handle all of the SOAP message processing, JAX-RPC handler processing, and web services configuration. Figure 8.1 shows the architecture of one running instance of GT4. In our setup we have using several physical machines with multiple running instances of GT4 on each machine.

8.3 The GridFTP Protocol

As previously stated in Chapter 4, base level operations on objects stored in the database are executed as local database transactions, and base level operations on objects stored on the local file system take advantage of Grid file management functionality, such as GridFTP [97]. In this section we briefly introduce the GridFTP protocol. GridFTP is a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. It is based upon the Internet FTP protocol, and it implements extensions for high-performance operation. GridFTP provides a fault tolerant implementation of FTP, to handle network unavailability and server problems. Transfers can also be automatically restarted if a problem occurs.

Some of the most noteworthy features of GridFTP are:

Parallel data transfer: GridFTP supports parallel data transfer through FTP command extensions and data channel extensions in order to improve the aggregate bandwidth on WANs.

Partial file transfer: Many applications (including our use case scenarios) may require the transfer of portions rather than complete files, which is particularly important for applications that require access to relatively small subsets of massive, object-oriented database files. Standard FTP supports the transfer of complete files or the transfer of the remainder of a file starting at a particular offset. GridFTP introduces new FTP commands to support transfers of subsets or regions of a file.

Support for reliable and restartable data transfers: Reliable transfer is essential for many applications and GridFTP incorporates fault tolerant features to handle transient network failures, server outages, etc. The FTP standard includes a basic feature for restarting failed transfers, but which is not widely implemented. The GridFTP protocol exploits these features and extends them to cover the new data channel protocol.

8.4 Re:GRIDiT Services

The Re:GRIDiT protocol is materialized in two distinct web services, the Transaction Manager (TM) Service and the Site Manager (SM) Service which build further functionality on top of the GT4 Java Core Services. They are implemented in Java and communicate to each other via SOAP messages. Each physical machine in the network contains at least one running instance of each of the two services. The transaction execution is migrated from one node to the other, when data needed during an operation invocation reside on a remote site. Thus, each Transaction Manager instance interacts with several Site Managers during its lifetime. Each Transaction Manager instance receives an automatically generated unique ID which enables the Transaction Managers to locate each other during their life time.

Transaction Manager Service The TM Service implements the Re:SYNCiT transaction protocol. Upon receiving a client transaction the TM Service checks whether it is an update or a read-only transaction. The separation between the two types of sites is handed over to the Site Manager Service. The TM Service executes operations on the appropriate sites, depending on the type of the operations: updates are executed optimistically on the update sites and reads on the read-only sites. For each update transaction the Re:SYNCiT protocol is run, consisting of of three phases (execution, validation, and commit), as described in Chapter 5. A background thread performs the serialization graph update and graph validation for cycles whenever a transaction is in the validation phase.

Site Manager Service The SM Service implements the Re:SYNCiT site protocol, described in detail in Chapter 5. For update transactions it detects local conflicts, which are stored in the local Derby database. Updates are propagated synchronously to the other update sites and the remote conflicts are returned. The remote conflicts are again stored in the local Derby database and given to each transaction during its validation phase. Backgrounds threads perform load balancing, take dynamic replica management decisions and propagate updates to read-only sites in a lazy fashion. For the implementation of the performance evaluation we have used the local CPU values only as load measurement. Programmatically querying for CPU usage is impossible using pure Java as so far there exists no API for this. A simple alternative was to use Runtime.exec() to call an external, platform-specific command like top, and parse its output. More sophisticated and reliable solutions can be accomplished by stepping outside Java and writing a few C code lines that integrate with the Java application via Java Native Interface (JNI). Every chosen time interval $\delta(t)$, the load is recalculated and if significant changes occur (significant differences in value with respect to the load values previously stored in the database) these changes are locally replaced and propagated together with replica synchronization.

As described in Chapters 5, 6 and 7, the Re:SYNCiT protocol relies on the communication between transactions and site. Its functionality is consequently implemented in both the TM and SM services. On the other hand, the Re:LOADiT and Re:FRESHiT protocols are exclusively implemented in the SM service.

8.5 Distributed Repositories

The distributed replicated repositories that are required in order to support replication have been implemented in a completely decentralized fashion. These repositories are: the replica catalog, the load repository, the freshness repository and the propagation queues. These repositories have been implemented as database tables in the local Derby databases. Although we have stressed the system we did not notice any performance degradation by using this approach. However, memory resident structures could also be used, if this proves to be the case. The idea of introducing such components is not new, similar approaches applied to peer-to-peer process management exist in the literature [163]. These local components contain global information, although depending on the nature of the site, the information may be partly outdated and partly replicated. In the case of update sites, this information is fully replicated together with replica synchronization. For the read-only sites this information is exchanged together with the update propagation.

8.6 Evaluation Setup

All the experiments have been conducted on up to 96 sites. All the machines have the following configuration. The sites are equipped with a Dual Intel®CPU 3.20 GHz processor and 5 GB RAM. As operating system we employed Ubuntu Linux 8.0.4 and Apache Derby as local database. The protocol has been implemented as Web services running inside a Java WS-Core 4.0.3 service container. Since Re:GRIDiT has been implemented using platform independent technologies, it has been successfully deployed on machines running various flavors of Unix, Windows and Mac OS X. As mentioned before, any off-the-shelf database can also be used instead of Derby.

8.7 Performance Metrics

In this section, the performance criteria used for the evaluation are introduced. In our setup clients issue requests, which are transactions consisting of multiple operations to be executed by the system. The operations are either update or read operations. Updates are synchronized among the update sites and later propagated to read-only sites. The time between the arrival of a request and the time that its operations start executing is called *waiting time* of a transaction, denoted *wt(t)*. The reasons why a transaction might wait to proceed could be high workload in the case of concurrent updates that arrive at update sites or the time requested for a refresh transaction to update a read-only site to the requested freshness level. The duration of the actual execution of a transaction is its *execution time*, denoted *et(t)*. The execution time of a transaction is the sum of the execution times of all its operations.

Throughout the evaluations we have used an average operation duration of 100 seconds, which the typical average duration for the transfer of a 1GB file using GridFTP [150]. Unless otherwise specified, we have used an average transaction size of five base level direct operations. This choice is not random. A typical example of a user transaction at the user level might look like:

$\mathrm{ReadCollection}(C)\mathrm{UpdateCollection} - \mathrm{AddDocument}(D)$.

Assume the simplistic case when document D is an environmental report which contains two information objects, a satellite image and a textual description of the methodology used to produce the data. At the same time, metadata about the image (author, date, time, satellite and region used to take the picture) are available and need be saved together with the image. This transaction will be automatically transformed at the middleware level into the following sequence of operations: "read collection", "insert image B on the file system", "insert metadata A (associated to B) into the database", "insert textual description C into the database", "replace recalculated index of the collection". These operations will be in turn mapped to base level operations that any Grid infrastructure is able to handle (such as GridFTP copy or SQL insert).

The following metrics have been used throughout the evaluations, unless otherwise specified:

Definition 8.1 ***(Runtime)*** *Let* t *be a client transaction. The overall runtime of a transaction* t*, denoted by* $runt(t)$ *is defined as:* $runtt(t) = wt(t) + et(t)$. □

The major performance metric used in the evaluation presented in Chapter 9 is the *throughput* of a system:

Definition 8.2 ***(Throughput)*** *Let* T *be a set of* n *client transactions (with* $|T| = n$*) that have been successfully committed within a time interval* τ*. The throughput of the system executing the transactions is:* $thp_{SYS}(T, \tau) = \frac{|T|}{\tau}$. □

We can extend this definition to include different type-specific throughput such as *query throughput* which calculates the throughput of read-only transactions and *update throughput* which is the throughput of update transactions.

9
Evaluations

This chapter presents the qualitative and quantitative results of the evaluation of each individual pillar of the Re:GRIDiT family. The setup used for the evaluation and the performance metrics have already been introduced in Chapter 8. Section 9.1 presents the analytical evaluation of the message complexity of Re:SYNCiT compared to other replication approaches, as well as the experimental results of the comparison between Re:SYNCiT and a typical eager replication scheme using 2PC in combination with S2PL. Section 9.2 discusses performance aspects of Re:LOADiT (implemented on top of Re:SYNCiT) compared with Re:SYNCiT (i.e., using a static replication scheme), for different workloads. Section 9.3 contains the evaluation of our Re:FRESHiT protocol for read-only transactions, and our query routing strategy. Finally, Section 9.4 sums up the main achievements of the evaluation.

9.1 Re:SYNCiT Evaluation

There are many concurrency control mechanisms in the literature, which can ensure that all executions are serializable (no matter what transactions are run). In practice, however, according to [76], the strict two-phase locking (S2PL) mechanism in combination with two-phase commit (2PC) is the dominant way to guarantee serializability for distributed transactions. Before discussing the experimental results of our evaluation, we provide an analytical evaluation of the Re:SYNCiT protocol for update transaction compared to some of the well-established replication mechanisms. As previously introduced in Chapter 3, it is a common approach for many replication protocols to use S2PL/2PC in order to guarantee serializability.

9.1.1 Analytical Evaluation

Message complexity is measured as the total number of messages required to commit or abort a transaction. Assume, for simplicity the case where a transaction consists of one operation only. Re:SYNCiT has a message complexity of $n + 2 * \mathsf{PRE}(T) + \mathsf{POST}(T)$, where n is the number of update sites, $\mathsf{PRE}(T)$ and $\mathsf{POST}(T)$ represent the number of pre- and post-ordered transactions of a given transaction T. In contrast to other approaches, the coordination in Re:SYNCiT relies on communication between transactions in order to reduce the communication between the sites. The number of messages exchanged between transactions that belong to the same graph is strongly influenced by the size of the graph, which in turn, is determined by the number of conflicts in the system. For such low conflict rates as we envision for this kind of applications, the communication between transactions is negligible compared to n. Well-established replication mechanisms such as DBSM [145], ROWAA [108] and S2PL/2PC exhibit a message complexity of $3n$, which may decrease to n if broadcast over IP is available. Other commit mechanisms, such as Paxos commit [93], are in general more fault-tolerant than 2PC but imply an even larger number of messages to be exchanged.

Most pessimistic concurrency control protocols, S2PL included, abort all concurrent conflicting transactions. Optimistic protocols, on the hand, suffer from the cascading aborts problem. In our approach we use partial rollback techniques to minimize the effects of cascading aborts. By communicating with each other, transactions involved in a cycle choose a victim which is totally compensated. The rest of the transactions involved in the cycle partially roll back until a point in time when the cycle disappears.

9.1.2 Estimating the Best Number of Replicas

We consider the availability of a data object to depend on the failure rate of the sites in the network. If a large number of sites is often unreachable, then a large proportion of data objects may become unavailable. The following function estimates the number of replicas n needed for a certain availability threshold.

Let n be the total number of replicas for a data object, p the average probability of a site to be up and α_D the required amount of availability for a data object D. Then the following holds: $\alpha_D = 1 - (1 - p)^n$.

Thus if we consider 80% availability of a data object as a minimal requirement, for a probability of 20% of sites to be up, the model recommends 7 replicas. For 10% probability of sites to be up, 15 replicas are needed; for 5%

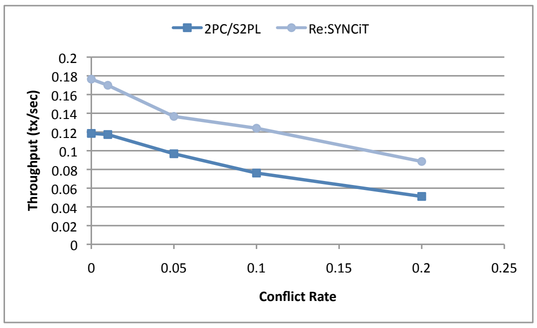


Figure 9.1: Throughput of S2PL vs Re:SYNCiT for different conflict rate values.

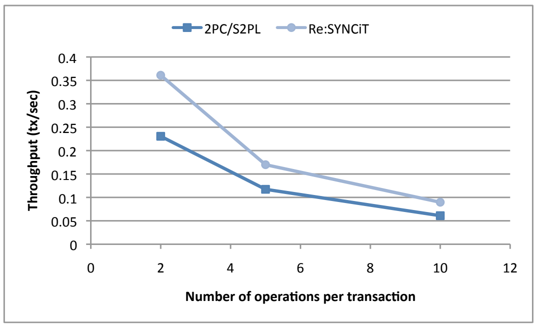


Figure 9.2: Throughput of S2PL vs Re:SYNCiT for different transaction lengths.

probability of sites to be up, 30 replicas, and for as low as 3% probability of sites to be up, 53 replicas.

9.1.3 Re:SYNCiT Transaction Throughput

In the following experiments, we wanted to learn if there are differences between Re:SYNCiT and 2PC/S2PL with respect to performance in terms of average throughput. We conducted several experiments varying the following parameters: different number of replica sites (8, 12, 16, 24, 32, 36 and 48 replicas), different conflict rates (1%, 5%, 10%, and 20%), and different transaction lengths (2, 5 and 10 direct operations respectively[1]). As calculated in Section 9.1.2, the choice of the different numbers of replica sites is not random, but motivated by the need for high availability of data (we have considered 80% availability as an acceptable value), while allowing very low average probabilities that the sites are up (roughly 20%, 10%, 5% up to 3%). The number of replica sites only considers the update sites – in addition there may be many more read-only sites in the network. We use the following transaction update rates: 10, 50, 100 and 500 milliseconds. This parameter strongly influences the number of transactions concurrently active at a certain point in time.

Our first experiment investigates the impact of the conflict probability on the throughput of Re:SYNCiT and 2PC/S2PL, respectively. Figure 9.1 shows the throughput of the protocols for various conflict rates, when the transaction length is kept constant at 5 operations per transaction and for a fixed number of replica sites. Even for a very high conflict rate of 20% Re:SYNCiT proves to perform better, although the difference in throughput is smaller

[1]However, with the transparent expansion of a transaction additional indirect operations are added.

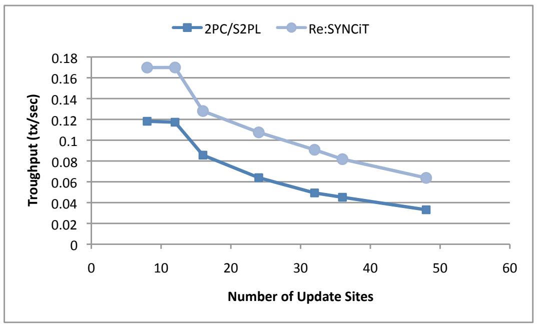


Figure 9.3: Throughput of S2PL vs Re:SYNCiT for different number of replica sites.

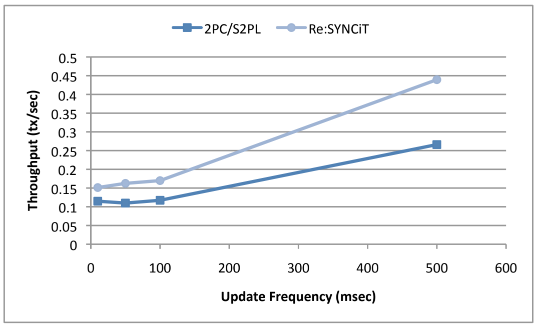


Figure 9.4: Throughput of S2PL vs Re:SYNCiT for different transaction update rates.

than for a conflict rate of 1%. Moreover, we investigated the impact of the transaction size on the throughput. We have run experiments with transaction sizes of 2, 5 and 10 operations respectively. Figure 9.2 shows the results of these experiments. The increase in transaction size leads to a decrease in the throughput of both S2PL and Re:SYNCiT, up to a point where the difference between the two becomes less significant. As it can be seen from Figure 9.3, an increase in the number of replica sites leads to a decrease in the throughput of both protocols, however, for low conflict rates (as we expect from our application scenario) Re:SYNCiT outperforms S2PL for any number of sites, showing that Re:SYNCiT is able to scale with the increasing number of replica sites. The same conclusions can be deduced from Figure 9.4 where an increasing update rate (more concurrently active transactions) leads to a consequent decrease in the throughput of both protocols with Re:SYNCiT still outperforming S2PL.

As observed from the experiments performed, Re:SYNCiT outperforms S2PL in terms of throughput. This is explained by the fact that in the case of S2PL transactions blocked because they cannot obtain a lock might themselves block subsequent operation invocations of other transactions. Re:SYNCiT, in contrast, allows the transactions in the same situation to optimistically continue. Figure 9.5 records the percentage of aborts for both protocols for different conflict rates, as well as the total runtime. These values represent the average values for measurements consisting of 10 runs of 500 transactions each. As expected, the total runtime for S2PL is, even in a conflict-free environment 33% higher than that of Re:SYNCiT, an overhead introduced by the locking management. This difference increases with the conflict rate, due to the blocking behavior of S2PL. In terms of number of aborts, for low conflict rates, both protocols behave the same way, whereas for medium conflict rates S2PL is slightly better than Re:SYNCiT. Even for

	2PC/S2PL			Re:SYNCiT		
Conflict rate	runt(t)	% aborts	# messages	runt(t)	% aborts	# messages
0.00	844.03	0.00	6040	566.70	0.00	5010
0.01	852.60	0.00	6150	588.61	0.00	5181
0.05	1033.53	0.02	6248	731.53	0.02	5289
0.10	1312.31	0.04	6866	804.93	0.06	5801
0.20	1929.43	0.74	7749	1127.58	0.25	6404

Figure 9.5: S2PL vs. Re:SYNCiT for different conflict rates.

high conflict rates, the partial rollback implemented for Re:SYNCiT makes it outperform S2PL. In terms of the number of messages exchanged, we can see that the increase in the conflict rate does not imply a linear increase in the number of messages exchanged since additional communication from the transactions is only required for transactions belonging to the same connected sub-graph. As already mentioned in our application scenario, conflicts are assumed to be rather infrequent in this type of applications. We can conclude that under these assumptions, the performance and scalability of Re:SYNCiT recommend it as a suitable protocol for this type of environment.

Serialization graph testing was used in the past only as a formal method to explain serializability theory, due to the runtime complexity of cycle checking. However, cycle checking is actually only then a problem when it is expensive compared to the operation execution cost. This might be the case for short living database transactions, but not in the context of potentially long-running transactions encompassing Grid services in distributed networks.

9.2 Re:LOADiT Evaluation

As a next step we have evaluated Re:LOADiT with support for dynamic replica placement and deployment against a static replication scheme. The goal of these evaluations is to verify the potential of the protocol, in terms of scalability and performance, compared to a protocol allowing only a semi-static replication scheme. For simplicity, we have used the site's CPU percentage as load measurement. The actual values used for the different load situations are depicted in Figure 9.6. These values have been chosen such that the intervals for a site promote or demote are comparable. Unless otherwise stated the measurements have consisted on runs of 100 transactions each. The conflict rate was set to 0.01 (since conflicts are assumed to be in-

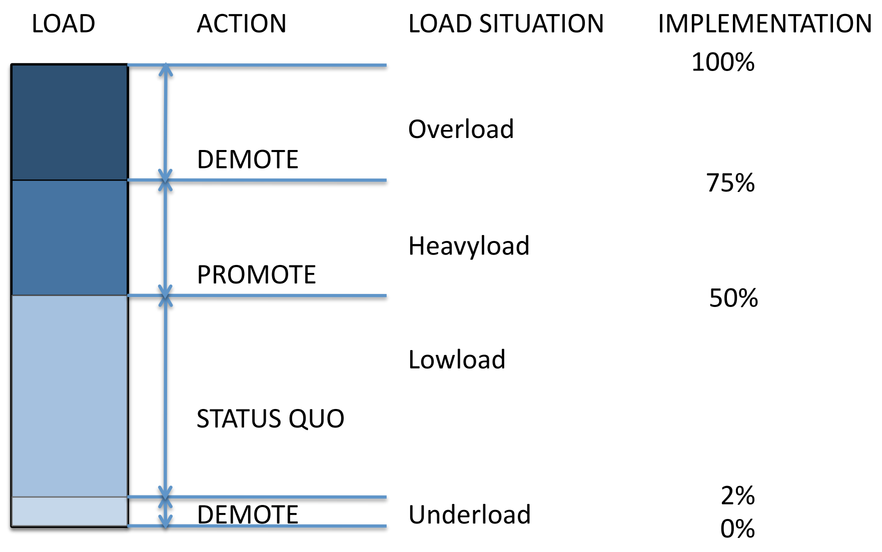
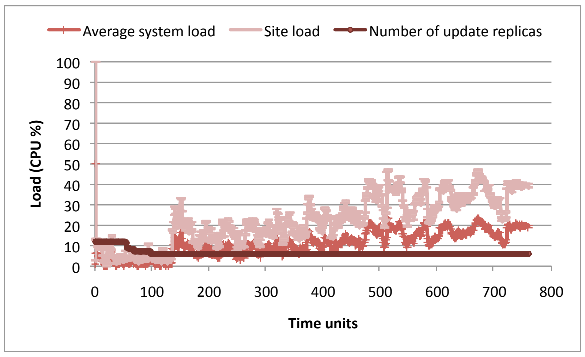


Figure 9.6: Load Thresholds in a Practical Evaluation.



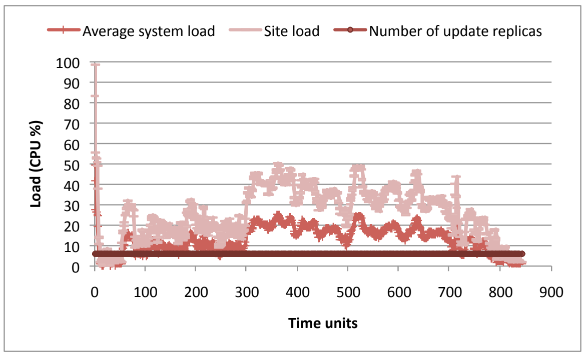


Figure 9.7: Load Variation in Time. Dynamic Setup (12 Initial Sites).

Figure 9.8: Load Variation in Time. Static Setup (6 Initial Sites).

frequent). Each transaction consists of 5 base level direct operations. The transactions were started sequentially, one after the other, with an update interval of milliseconds, such that as many transactions as possible are active at the same time.

9.2.1 Re:LOADiT Load Variation in the Presence of Active Transactions

In order to evaluate the performance of our protocol we have conducted several experiments that evaluate the load variation in the presence of active transactions at the sites. We have recorded the mean local load variations, the mean system loads and the evolution of the number of replicas in time (in

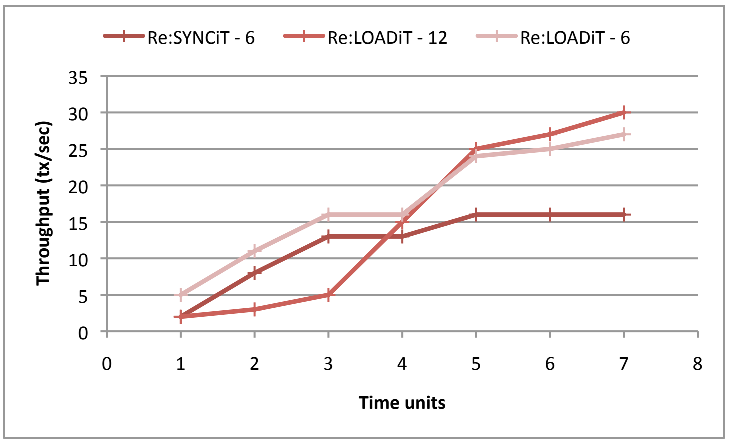


Figure 9.9: Re:LOADiT versus Re:SYNCiT Throughput Variation.

a dynamic setting). As observed from Figure 9.7 for an initial number of 12 update sites, the average number of replicas in the system tends to remain stable at an minimum of 6. We have compared these results with the static replication protocol for which we used 6 update sites. The average system load shows more fluctuations in the dynamic case (due to the dynamic replica management overhead), and on average, the local site loads show a less than 5% increase with respect to the static case. In this case, the static setting of 6 update replicas has been chosen to match the optimal minimum which is reached by the dynamic setting; therefore an unfortunate choice of less than optimal number of replicas in the static setting produces non-negligible load differences with respect to the dynamic setting.

9.2.2 Re:LOADiT Transaction Throughput

Another means of evaluating the performance of the Re:SYNCiT and Re:LOADiT protocols is by comparing their throughput, calculated as the number of transactions committed within a time interval (in this case 20 seconds). In this measurement, both protocols with an initial setup of 6 sites and the dynamic protocol with an initial setup of 12 sites have been compared. As in the previous cases, the dynamic replication protocol has stabilized the number of update sites at a minimum of 6. As it can be seen from Figure 9.9, the throughput of Re:SYNCiT with 6 replicas is initially higher than that of Re:LOADiT with 12 initial replicas and comparable to the one of Re:LOADiT with 6 initial replicas. However, both dynamic settings stabilize at a lower number of replicas and soon outperform the static one. The reason for this behavior is that transactions begin the commit much sooner in time in the dynamic case than in the static one. The rationale behind it is

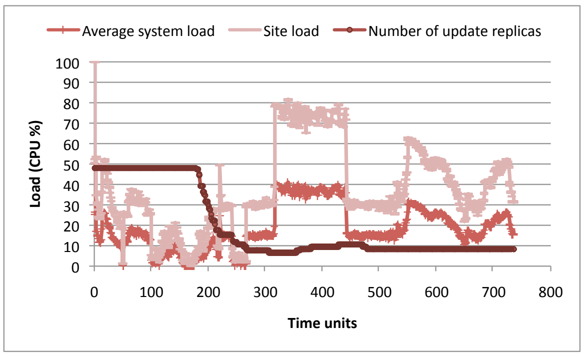


Figure 9.10: Load Variation in Time. Dynamic Setup (48 Initial Sites).

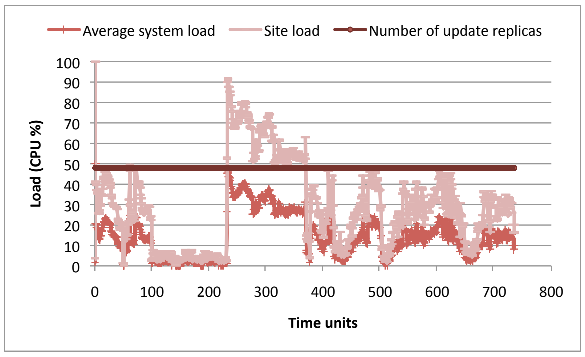


Figure 9.11: Load Variation in Time. Static Setup (48 Initial Sites).

that the Re:SYNCiT requires more time to synchronize the update to a higher (and constant) number of update sites in the case of 6 static sites, therefore the transaction duration is higher. It can also be observed that initially the throughput in the dynamic setting is smaller than in the case of the static setting with 6 update sites, due to the extra load imposed by the demote of the unnecessary update sites. Nevertheless, the throughput of the dynamic setting is increasing and outperforms the throughput of the static setting with 6 update sites, due to the selection of the best replica sites in the dynamic case.

9.2.3 Re:LOADiT Load Variation in the Presence of Active Transactions and Additional Load

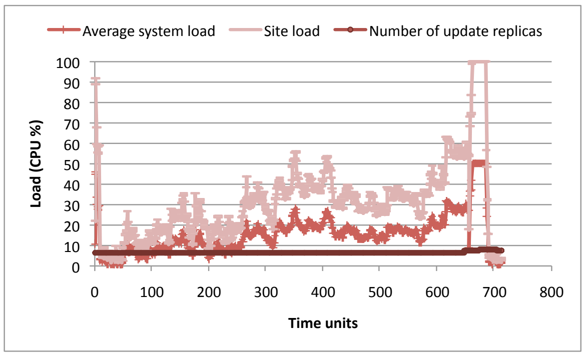


Figure 9.12: Load Variation in Time. Dynamic Setup (6 Initial Sites).

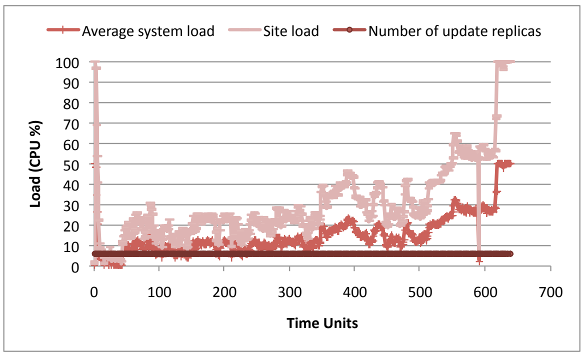


Figure 9.13: Load Variation in Time. Static Setup (6 Initial Sites).

As it can be seen from Figures 9.7 and 9.8, the distributed concurrency control of active running transactions and the replica management hardly drive the CPU load within the *heavyload* level and never in the *overload*

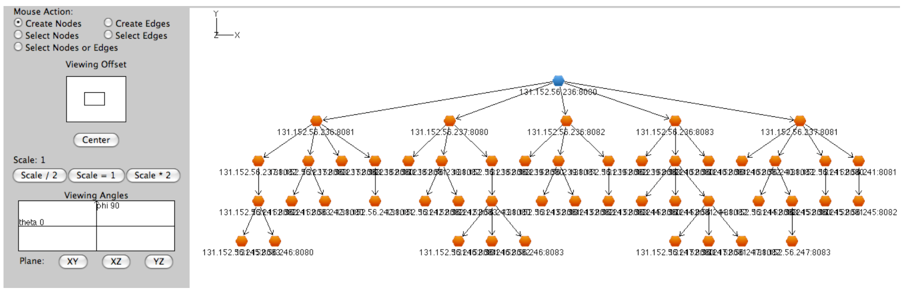


Figure 9.14: Hierarchical Topology of the Network.

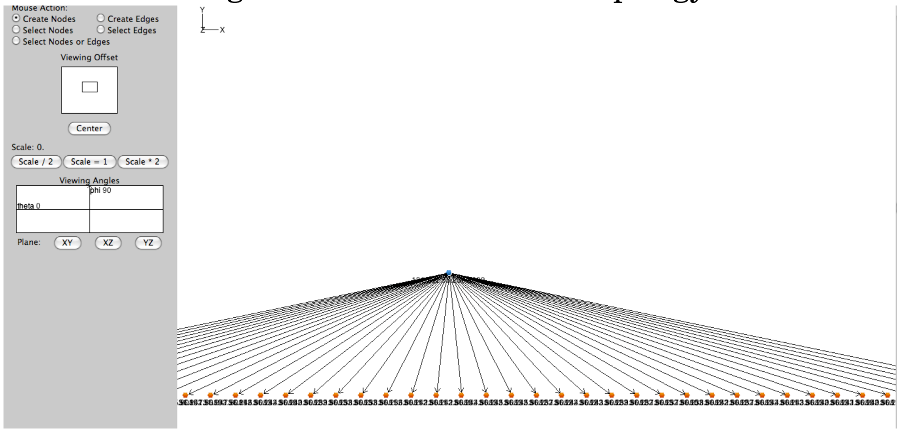


Figure 9.15: Flat Topology of the Network.

level. In order to observe the system's behavior under heavier load stress we have tested the protocols in a setting using active transaction and artificial load variations (that mimic the behavior of additional read operations). These load variations introduce dynamic changes in the system which lead to promote and demote situations. In each of the cases the load has been maintained stable for several minutes in order to give the system enough time to react to the changes. The results can be seen in Figures 9.10 and 9.11 where an initial setup of 48 update sites is compared and Figures 9.12 and 9.13 where an initial setup of 6 update sites is compared. It is clear that while both protocols are subject to the same load levels Re:LOADiT is able to better cope with the varying load situations and consequently promote and demote sites as needed.

9.3 Re:FRESHiT Evaluation

Our next series of evaluation shows the performance of Re:FRESHiT in different network topologies and different routing strategies.

We compared two network topologies (as presented in Figures 9.14 and 9.15) in order to evaluate the advantages of our proposed architectural structure:

- In the first setting we propose a tree structure (as introduced in Chapter 4), where sites are classified into three levels: Level 1 update sites, Level 2 read-only sites (with higher freshness levels) and Level 3 read-only sites (with staler copies of data). The tree structure used in the evaluation is presented in Figure 9.14. We call this setting Re:FRESHiT-TREE.

- In the second setting, we evaluate a two levels structure, where the Level 3 read-only nodes are eliminated completely. The tree structure used in the evaluation is presented in Figure 9.15. We call this setting Re:FRESHiT-FLAT. This configuration has also been adopted in [10].

Updates that occur at update sites are continuously propagated to read-only sites. Hence, in addition to refresh transactions, which occur on demand (as a consequence of user queries), changes are bulked into propagation transactions and applied to the sites whenever possible. For this experiment we have used a ratio of concurrent update transactions to queries of 1:10. We considered the update rate at update sites as the unit size of bulked updates. The update rate used in the experiments was 100 updates per second. Each update and read-only transaction consists of 5 base level operations each. In the comparison we varied the freshness degree of data requested by a query. The tests were repeated for client requests for data with freshness levels within the following freshness intervals: [0.5;1], [0.7;1] and [0.9;1]. Freshness functions belong to a given interval and are mapped to transaction timestamps.

Our second series of experiments evaluated the advantages of query re-routing along the tree structure. We compared two routing strategies:

- In the first setting we re-route user queries up or down the tree structure according to freshness and load, according to Algorithm 8. We refer to this general setting as Re:FRESHiT.

- In the second setting we modify the Re:FRESHiT protocol to force requests to be processed locally. This means that requests are no longer re-routed along the tree structure. In case a data object on a site does not have the required freshness level, a refresh transaction is initiated on demand. We refer to this setting as FORCE-Refresh.

Our third series of experiments evaluated the advantages of tree rotation whenever refresh transactions occur. We compared two refresh strategies:

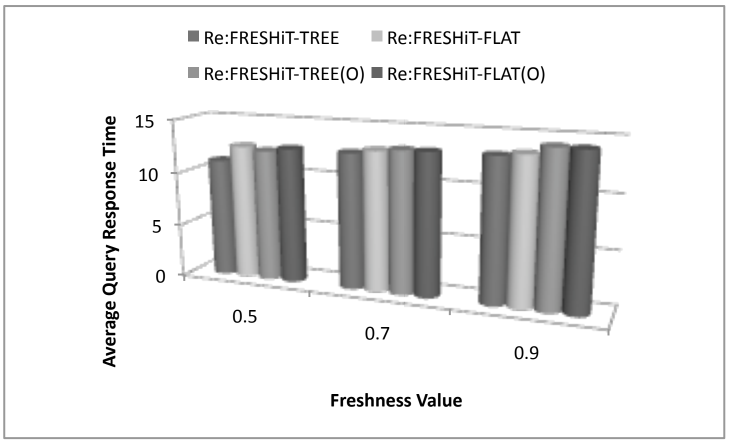


Figure 9.16: Average Query Response Time for Different Network Topologies.

- In the first setting we dynamically rotate the tree whenever a query that arrived at a site cannot be serviced at that site and therefore requires a refresh transaction, according to Algorithm 8. We refer to this general setting as Re:FRESHiT.
- In the second setting we modify the Re:FRESHiT protocol to refresh the entire tree structure whenever a query that arrived at a site cannot be serviced at that site and therefore requires a refresh transaction. In case a data object on a site does not have the required freshness level, and the request cannot be re-routed, the refresh transaction refreshes all the predecessors of that site in the tree hierarchy. We refer to this setting as Re:FRESHall.

The latter two strategies have been tested using the Re:FRESHiT-TREE network topology.

9.3.1 Re:FRESHiT Network Structure

Our first experiments show how the network topology and the freshness requirements influence the query response time. We used queries with freshness requirements within three different intervals: [0.5;1], [0.7;1] and [0.9;1]. The results are depicted in Figure 9.16. The results show the advantages of the tree topology versus the flat topology. By reducing the number of Level 2 sites in comparison to Re:FRESHiT-FLAT, we reduce the propagation time, which explains the increase in performance. Furthermore, since we assume a high workload, propagation transactions are slower than refresh transactions, however they are still able to keep sites fresh enough for queries with lower freshness requirements. By taking advantage of the tree structure,

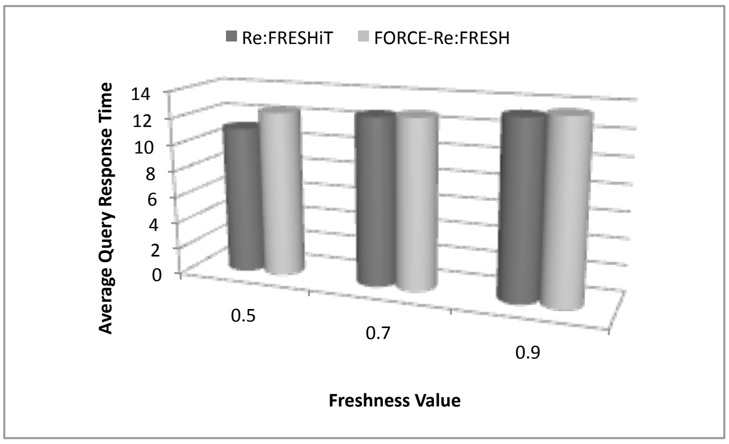


Figure 9.17: Average Query Response Time for Different Route Strategies.

queries have better chance of finding a site that satisfies a lower degree of freshness, which can be seen by the 15% increase in query response time for Re:FRESHiT-TREE.

9.3.2 Re:FRESHiT Query Routing

Our second set of experiments shows how the query re-routing and the freshness requirements influence the query response time. Again we used queries with freshness requirements within three different intervals: [0.5;1], [0.7;1] and [0.9;1]. The results are depicted in Figure 9.17 and show the advantages of Re:FRESHiT versus FORCE-Refresh, especially when processing queries with a lower degree of freshness. By routing the queries within the tree structure we reduce the time required by the refresh transactions. Since the freshness is monotonically decreasing within a tree, a site is able to properly route a query upwards in the tree if the client request has a higher timestamp than the local one, or downward in the tree if it has a lower timestamp. The difference in performance is reduced for queries with a higher freshness levels, as in this case the refresh transactions would still be needed. Nevertheless Re:FRESHiT shows an almost 20% increase in the query response time for user queries with lower freshness levels.

9.3.3 Re:FRESHiT Refresh Strategies

Our third set of experiments show how the tree dynamic structure and the freshness requirements influence the query response time. Again we used queries with freshness requirements within three different intervals: [0.5;1], [0.7;1] and [0.9;1]. The results depicted in Figure 9.18 show the ad-

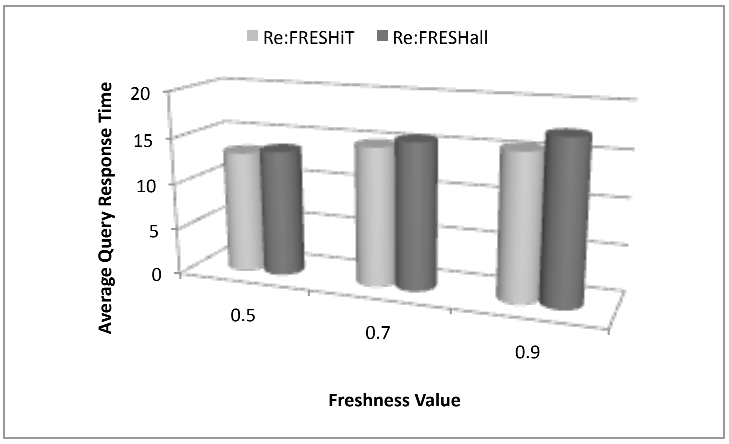


Figure 9.18: Average Query Response Time for Different Refresh Strategies.

vantages of Re:FRESHiT versus Re:FRESHall, when processing queries with all degrees of freshness. By dynamically rotating the tree structure we reduce the time required by the refresh transactions, as in the case of Re:FRESHiT they are applied at one site only rather than at a subset of the sites along a certain portion of the path as in the case of Re:FRESHall. Using our routing strategy, queries are still routed by taking advantage of the monotony of the freshness. However, by splitting a potentially long path in the tree in several sub-trees of shorter paths we ensure that data with higher freshness levels are also available and that the re-routing of queries takes less time. Re:FRESHiT shows an up to 30% increase in the query response time for user queries with all freshness levels.

9.4 Summary

One the main goals of this thesis has been the implementation and evaluation of an integrated approach to dynamic decentralized replication management in data Grids with freshness and correctness guarantees. In this chapter we have presented the evaluation results of Re:GRIDiT in a realistic data Grid setting. Each pillar of the Re:GRIDiT protocol has been tested and evaluated separately in a realistic environment.

Our first series of experiments concentrated on the Re:SYNCiT protocol and shows its advantages compared to S2PL. Re:SYNCiT exhibits increased performance and scalability when compared to S2PL. For this purpose we have used different conflict rates, different number of update sites, update rates or number of operations per transaction.

The second step has been to envisage and test a dynamic approach that uses a combination of load, freshness and host proximity criteria to balance the load between replicas. As it can be seen from Figures 9.10 and 9.12 dynamic changes in the load determines when new replicas need to be acquired or released. Moreover, replica acquisition does not impose a significant overhead. Figure 9.7 reflects the overhead in replica acquisition with respect to the static protocol presented in Figure 9.8 which is shown to be at approximately 5%.

Our third and last step towards extending the Re:GRIDiT system to a fully mature replication management system has been the implementation of the Re:FRESHiT protocol. Re:FRESHiT proposes a dynamic tree topology and a clever routing mechanism that improves the users access to data, while at the same time allowing users to specify their own freshness requirements. The evaluation of our replication protocol and the comparison in different settings with different network topologies and routing strategies has shown a 20 – 30% increase in the query response time for user queries with different freshness levels.

All Re:GRIDiT protocols allow data to be arbitrarily partitioned across sites and do not require any global component in the network. Summarizing, the main achieved goals while evaluating the Re:GRIDiT protocols are:

- **Dynamic**: replicas can be created and deleted dynamically when the need arises. Dynamic changes in the tree structure allow flexible and efficient query routing along the tree structure.

- **Efficient**: replicas are created in a timely manner and with a reasonable amount of resources. Clever refresh and routing strategies ensure an increased performance for queries with different freshness levels.

- **Flexible**: replicas are able to join and leave the Grid when needed. Our dynamic replication protocol allows replicas to join and leave the replication scheme as long as a minimum number of replicas is present in the system. This limit is application dependent since different application scenarios might have different needs.

- **Replica Consistency**: in an environment where updates to a replica are needed, different degrees of consistency and update frequencies should be provided. In all measurements the Re:SYNCiT distributed and optimistic concurrency control protocol is enabled. At the same time relaxing freshness requirements for read-only sites still guarantees consistency.

- **Scalable**: the replication system is able to handle a large number of replicas and simultaneous replica creation. We have tested our protocol on a setup consisting of up to 48 update sites and up to 48 read-only sites.

10

Data Grids and Data Clouds - Synergies and Opportunities

Cloud computing has recently received considerable attention both in industry and academia [8]. There may be a vast array of reasons as to why an individual or business might use Cloud computing. Some reasons include: scalability, flexibility, reliability, fast setup, affordable solution, environmentally more efficient and the list may continue. Cloud computing increases capacity or adds capabilities on the fly without having to purchase and maintain physical hardware as well as the space to store it reduces overhead costs, training new personnel, or licensing new software. In this chapter we introduce Cloud computing and make a comparison between data Grids and Cloud computing from the point of view of data management. This allows us to evaluate the feasibility of applying our approach to other distributed environments, and in particular Cloud computing.

10.1 Introduction to Cloud Computing

Cloud-based computing allows customers to rent hardware and/or software resources, thus being freed from significant investments in building up and maintaining computing centers in-house by outsourcing their complete ICT infrastructure. Resources are made available according to Quality-of-Service (QoS) guarantees which are negotiated between provider and customer. Depending on the type of resources which are made available to customers and the services which have been negotiated, there is a distinction between Infrastructure as a Service (IaaS) [3], Platform as a Service (PaaS), and Software as a Service (SaaS). Providers of Cloud-based services usually maintain different distributed data centers. This allows to dynamically adapt the resources provided for a particular customer based on their current needs (within the QoS agreement that has been negotiated).

A core challenge in the context of Cloud computing is the management of very large volumes of data. This is completely independent of the type of resource which is shared in the Cloud – databases are either directly visible and accessible to customers as part of the infrastructure/platform, or are hidden behind service interfaces. In terms of data management, QoS guarantees mainly encompass a high degree of availability. For providers of Cloud services, this means that data need to be partitioned and replicated across different data centers. Although traditional high throughput OLTP applications are most likely not to become the predominant applications hosted in a Cloud environment [1], replicated data management nevertheless needs to take into account updates which are performed on replicated data (either directly or via service calls). Replicated data management in the context of concurrent updates to different replicas can be addressed either by using well-established protocols such as strict two-phase locking (2PL) in combination with two-phase commit (2PC) [95], or by relaxing ACID properties to increase the overall performance and throughput of the system. The latter is applied in several of today's Cloud environments (e.g., PNUTS [62]).

10.2 Distributed Data Management: Cloud vs. Grid

Data Grids and Cloud Data Management share similar objectives. However, the development of the Grid and of the Cloud have only been loosely coupled for several reasons. First, they both focused on specific user communities: scientific communities (eScience) in case of the Grid vs. the outsourcing of ICT services for commercial customers in case of the Cloud. Second, both environments have different origins: the main driver for the Grid has been the High Energy Physics community (other eScience communities have adopted the Grid rather recently), while the proliferation of the Cloud has been dominated by large providers of IT services that already had the necessary computing resources (data centers) in place and were heading towards a more optimal utilization of their capacities. Third, the initial requirements which have been addressed were different. In the data Grid, first solutions have focused on the controlled sharing of files within Virtual Organizations (VOs). Data management at a granularity finer than files, replication management and updates have only very recently been put on the list of requirements due to novel eScience applications (e.g., earth observation, health care, etc.). For Cloud-based environments, analytical data management has been

	Cloud Data Management	Existing Data Grids
Distribution	Few data centers	Many grid nodes (i.e., data centers)
Environment	Homogeneous resources	Heterogeneous Grid nodes
Data Access	SQL interface, possibly also semantically rich operations	Read and write files
Replication	QoS (availability)	QoS (availability)
Replication Granularity	Fine-grained: individual tuples (multi-tenancy)	Coarse-grained: files
Updates	No traditional OLTP load, but concurrent updates on replicated data	Read-only access (but novel applications also demand updates)
Global Control	Global repositories – potential bottleneck (scalability)	Global repositories – potential bottleneck (scalability)
Global Correctness	Relaxation of ACID properties	– (no updates)
Dynamic Changes	Needed in order to support horizontal scaling	Only static replication
Data Freshness	Considered as a consequence of relaxed ACID properties	– (no updates)

Figure 10.1: Cloud Data Management vs. Data Grids: a Comparison

identified as the predominant application [1]. However, in the presence of QoS constraints that need to be met by Cloud service providers, data need to be replicated across data centers. Although the percentage of updates will be rather low compared to traditional OLTP settings, Cloud Data Management nevertheless needs to provide correct and consistent data management in the presence of conflicting updates. Therefore, despite the initial separation between data Grids and Cloud Data Management, the requirements both environments need to address more and more converge.

According to [60], the Cloud needs to meet the following requirements:

Multi-tenancy: *A Cloud service must support multiple, organizationally distant customers.* Multi-tenancy is also an important requirement in the Data Grid. However, support for VOs and thus for different users/customers within the same distributed infrastructure is already integral part of most Grid middleware systems.

Elasticity & Resource Sharing: *Tenants should be able to negotiate and receive resources on-demand. Spare Cloud resources should be transparently applied when a tenant's negotiated QoS is insufficient [142].* Similarly, the resources made available to a VO in a Grid should be dynamically adapted to its current needs. Therefore, QoS-based resource negotiation and allocation has recently become an important topic also in the Grid community.

Horizontal Scaling & Security: *It should be possible to add Cloud capacity in small increments, transparent to the tenants. A Cloud service should be secure such that tenants are not made vulnerable because of loopholes in the Cloud.* Similar requirements can also be found in the Grid.

Metering: *A Cloud service must support accounting that reasonably ascribes operational and capital expenditures to each of the tenants of the service.* As Data Grids have their origin in scientific communities and operate on resources which are contributed to by the member institutions of VOs on a voluntary basis, this aspect has not yet been in the main focus of Grid environments.

Availability: *A Cloud service should be highly available.* This requirement can also be found in the Grid. In addition, as services need the (local) presence of the data they access, availability should be extended also to the underlying data sources by means of replication.

Operability: *A Cloud service should be easy to operate.* In general, this requirement also holds for the Grid. However, due to the heterogeneous environment in which Grids can be deployed, some current Grid middleware solutions are rather limited in that regard. But with the proliferation of Service Grids, this limitation more and more diminishes.

Figure 10.1 summarizes the relationship between Cloud Data Management and Data Grids. The table shows that differences between both fields still exist. However, Re:GRIDiT which follows a novel approach to data management in the Grid by making use of and extending protocols that have originally been devised for database clusters, can be considered a major contribution to the convergence of both areas.

10.3 Re:GRIDiT for the Cloud

In order to show the potential of Re:GRIDiT for Cloud Data Management, we have evaluated the system in realistic Cloud settings. Since Cloud-based environments typically contain less but more powerful resources than a Grid (e.g., several data centers of a Cloud service provider rather than a large machines with free capacities within an eScience community), we have run experiments with up to 12 sites. The evaluation results have been presented in [184] and have shown that Re:GRIDiT, a protocol that has originally been devised for replicated data management in the Grid, is very suitable also for Cloud Data Management.

	Cloud Data Management	Re:GRIDiT
Distribution	Few data centers	Scales up to many nodes (Grid-scale)
Environment	Homogeneous resources	Re:GRIDiT on heterogeneous nodes
Data Access	SQL interface, possibly also semantically rich operations	operations for mutable / immutable data; collections & documents
Replication	QoS (availability)	QoS (availability)
Replication Granularity	Fine-grained: individual tuples (multi-tenancy)	Several granularities possible: files, realations, partitions, tuples
Updates	No traditional OLTP load, but concurrent updates on replicated data	Update and read-only nodes
Global Control	Global repositories – potential bottleneck (scalability)	Completely distributed, no global component ➔ scalability
Global Correctness	Relaxation of ACID properties	Eager replication among update nodes (➔ serializability)
Dynamic Changes	Needed in order to support horizontal scaling	Dynamic replica creation (update nodes vs. read-only nodes)
Data Freshness	Considered as a consequence of relaxed ACID properties	Different freshness levels (mutable data) and versions (immutable data)

Figure 10.2: Cloud Adaptability for Re:GRIDiT

Figure 10.1 shows how Re:GRIDiT can be seamlessly applied to Cloud Data Management. The table emphasizes how Re:GRIDiT is capable of bridging the gap between Cloud data management and data Grids.

10.4 Summary

Although having started as specialized solutions for different communities and with different sets of requirements, Cloud Data Management and data Grids are more and more converging. In this chapter, we have analyzed the commonalities and differences that still exist between both areas.

The Re:GRIDiT system provides advanced data and replication management in the Grid [186, 187] and follows a truly distributed approach to replication management in the Grid by bringing together approaches from replication management originally developed for database clusters [10, 157], and distributed transaction management that does not rely on a global coordinator [101]. A most important feature of Re:GRIDiT is that it has been designed to be independent from any underlying Grid middleware. Thus, Re:GRIDiT can also be seamlessly deployed in other environments like the Cloud and performs very well in such environments.

11

Conclusions And Outlook

The Grid integrates distributed computational and data resources to create a single virtual resource which provides its users with potentially unlimited processing and on-demand data storage power [125]. In contrast to first Grid applications which were developed for physicists, the Grid no longer exclusively targets scientific applications working with mostly read-only data. Data Grids are providing a cutting-edge technology of which scientists and engineers are trying to take advantage by pooling their resources in order to solve complex problems and have known intensive developments over the past years. Basic middlewares are available and it is now time for the developers to turn toward specific application problems. The emergence of new application domains such as digital libraries, earth observation or eHealth, requires data management schemes suitable for distributed data in a very-large scale Grid. To develop such management schemes we resolve data availability, data consistency, and data versioning issues for the data Grid environment.

In order to succeed in our three-fold approach to dynamic replication in data Grids with freshness and correctness guarantees, the first problem we were faced with was the coordination of distributed update transactions with replicated Grid data, thus solving the problem of transaction synchronization and replica management for the more difficult and more general case of update sites. The Re:SYNCiT protocol for the eager replication of update sites has successfully achieved this goal. The Re:LOADiT support for dynamic replication has ensured a fine load balancing between other replicas, using a dynamic model which takes several parameters into account. Re:LOADiT is based on the optimistic Re:SYNCiT protocol and avoids performance degradation due to locking schemes. At the same time, relaxed freshness and consistency models can be used, when applications that can survive with lower levels of freshness. Re:FRESHiT allows applications to specify a desired freshness level as a quality of service parameter. The notion of freshness is also used to choose the best replica for a particular situation. Furthermore updates are propagated to read-only sites in a lazy manner. As an integrated approach, Re:GRIDiT has proven to be an efficient replica maintenance. Since synchronous replication is expensive, a combination of eager and lazy replication mechanisms with consistency and efficiency guarantees is used.

In more detail, the Re:GRIDiT family of protocols has achieved the following goals:

Re:SYNCiT synchronizes updates to several replicas in the Grid in a distributed way. Our approach assumes no global coordinator: we enforce globally serializable schedules in a completely distributed way without relying on a central coordinator with complete global knowledge. Last but not least, we support a flexible data model in which we distinguish between mutable and immutable data objects. Mutable data objects can be updated. Immutable data objects, on the other hand, cannot be modified; once created they are kept until deleted, but several versions of the same immutable data object may exist. To the best of our knowledge this distinction between data objects has not been made in any available protocol, yet it is a straightforward consequence of the nature of many Grid applications.

Re:LOADiT approaches replica deployment and management in a dynamic way. In our system user requests can be directed to and executed by any replica, and the destination of a request is determined by the following parameters: load, freshness or network distance to the replica. We distribute data objects among several replicas to raise throughput and move frequently used/heavy accessed data objects to relatively inactive replicas, where they do not compete against each other for resources, and requests can be handled faster. Based on a combination of local load statistics, proximity and data access patterns, Re:LOADiT dynamically adds new replicas or removes existing ones without impacting global correctness.

Re:FRESHiT allows read-only clients to state how up-to-date their data should be. Users may demand a certain freshness level or a certain version and this includes the special case where users always want to work with up-to-date data. Re:FRESHiT supports the freshness-aware routing of queries in the Grid and also takes into account the sites' local load for replica selection without relying on any central component. In parallel, updates are propagated from the update sites to the read-only sites along the proposed site hierarchy in a consistent manner.

The contribution of this thesis is the design, implementation and evaluation of the Re:GRIDiT protocols for dynamic replication management in a data Grid with freshness and correctness guarantees. Our extensive evaluations have proven that Re:GRIDiT is a mature replication management approach that exhibits performance and scalability when applied at Grid scale.

While the work presented in this thesis has achieved its goal, there are still related research problems that warrant further consideration and investigation.

Middleware scalability: We have shown how our routing techniques can lead to significant performance improvements and that these results provide a good scalability with environments of up to 96 sites (48 update sites and 48 read-only sites). Our experiments support our conclusions on the investigated Grid size. More general statements would require the implementation of our protocol on a larger scale, using sites connected via a WAN (instead of just local resources) and using larger data sets as found in many eScience applications.

Further application areas: We have provided an introduction to Cloud computing and presented data management issues present in these environments. So far, we have empirically evaluated the feasibility of applying Re:GRIDiT to cloud environments. Possible extensions would include exploring the challenges posed by transferring our protocol into the Cloud and the support for more fine-grained replication or a combination of several protocols with different Quality-of-Service guarantees in order to support multi-tenant applications.

Bibliography

[1] D. Abadi. Data Management in the Cloud: Limitations and Opportunities. *IEEE DE Bulletin*, 32(1):3–12, 2009.

[2] J. Abawajy. Fault-Tolerant Scheduling Policy for Grid Computing Systems. *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004.

[3] A. Aboulnaga, K. Salem, A. Soror, U. Minhas, P. Kokosielis, and S. Kamath. Deploying Database Appliances in the Cloud. *IEEE DE Bulletin*, 32(1):13–20, 2009.

[4] R. Acharya, R. Wasserman, J. Stevens, and H. C. inojosa. Biomedical imaging modalities: a tutorial. *Computerized Medical Imaging and Graphics*, 19:3–25(23), 1995.

[5] D. Agrawal and A. E. Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In *VLDB*, pages 243–254, San Francisco, USA, 1990.

[6] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases (extended abstract). In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–172, New York, NY, USA, 1997. ACM.

[7] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: version control and concurrency control. *SIGMOD Rec.*, 18(2):408–417, 1989.

[8] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. The Claremont Report on Database Research. *Commununication of the ACM*, 52(6):56–65, 2009.

[9] F. Akal, H. Schuldt, and H.-J. Schek. Grid-Enabled Data Replication for Digital Libraries with Freshness and Correctness Guarantees. In *3rd VLDB Workshop on Data Management in Grids, Vienna, Austria*, 2007.

[10] F. Akal, C. Türker, H.-J. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In *VLDB*, pages 565–576, 2005.

[11] R. Akbarinia, E. Pacitti, and P. Valduriez. Data currency in replicated DHTs. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 211–222, 2007.

[12] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H.-J. Schek, and G. Weikum. Unifying concurrency control and recovery of transactions. *Inf. Syst.*, 19(1):101–115, 1994.

[13] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[14] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: are these mutually exclusive? In *SIGMOD*, pages 484–495, New York, NY, USA, 1998. ACM.

[15] A. Andrzejak, S. Graupner, V. Kotov, and H. Trinks. Algorithms for Self-Organization and Adaptive Service Placement in Dynamic Distributed Systems. Technical report, HP, 2002.

[16] J. Annis, Y. Zhao, J. Voeckler, M. Wilde, S. Kent, and I. Foster. Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey. In *The 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, November 2002.

[17] S. Antony, D. Agrawal, and A. E. Abbadi. P2P systems with transactional semantics. In *EDBT*, pages 4–15. ACM, 2008.

[18] M. Assante, L. Candela, D. Castelli, L. Frosini, L. Lelii, P. Manghi, A. Manzi, P. Pagano, and M. Simi. An Extensible Virtual Digital Libraries Generator. In *Proceedings of the 12th European Conference on Research and Advanced Technology for Digital Libraries, ECDL 2008*, pages 122–134, September 2008.

[19] M. Assante and L. Frosini. Extensible Digital Library User Portals for e-Infrastructures. In *Proceedings of the 1st Workshop on Very Large Digital Libraries, VLDL2008*, September 2008.

[20] C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *J. ACM*, 36(2):230–269, 1989.

[21] C. Beeri, P. A. Bernstein, N. Goodman, M. Y. Lai, and D. E. Shasha. A concurrency control theory for nested transactions (preliminary report). In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 45–62, New York, NY, USA, 1983. ACM.

[22] W. Bell, D. Cameron, R. Carvajal-Schiaffino, P. Millar, and K. Stockinger. Evaluation of an Economy-Based File Replication Strategy for a Data Grid. *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2002.

[23] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24:1–10, May 1995.

[24] P. Bernstein and E. Newcomer. *Principles of transaction processing: for the systems professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[25] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[26] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst.*, 9(4):596–615, 1984.

[27] P. A. Bernstein and N. Goodman. Serializability theory for replicated databases. *J. Comput. Syst. Sci.*, 31(3):355–374, 1985.

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

[29] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Mobile Data Management*, pages 3–14, 2001.

[30] M. L. Bote-Lorenzo, Y. A. Dimitriadis, and E. Gómez-Sánchez. Grid Characteristics and Uses: A Grid Definition. In *Across Grids 2003*, pages 291–298, 2004.

[31] P. Bradley, J. Gehrke, R. Ramakrishnan, and R. Srikant. Scaling mining algorithms to large databases. In *Commun. ACM*, volume 45, pages 38–43, 2002.

[32] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *EDBT*, pages 97–108, 1999.

[33] Y. Breitbart and H. F. Korth. Replication and consistency: being lazy helps sometimes. In *PODS*, pages 173–184, 1997.

[34] Y. Breitbart, P. Olson, and G. Thompson. Database Integration Issues in A Distributed Heterogeneous Database System. *Multidatabase Systems: An Advanced Solution for Global Information Sharing*, 1993.

[35] Y. Breitbart and L. Tieman. *ADDS - Heterogeneous Database System, in Distributed Data Sharing Systems*. F. Schreiber and W. Litwin, North Holland, 1985.

[36] Y. Breitbart, R. Vingralek, and G. Weikum. Load Control in Scalable File Structures. *Distributed and Parallel Databases*, 4(1), 1996.

[37] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM.

[38] Business Transaction Protocol Primer, An OASIS Committee Supporting Document. http://www.oasis-open.org/committees/download.php/2077/BTP_Primer_v1.0.20020605.pdf.

[39] K. Budati, J. Sonnek, A. Chandra, and J. Weissman. RIDGE: Combining Reliability and Performance in Open Grid Platforms. In *Proceedings of the 16th international symposium on High performance distributed computing*, pages 55–64, New York, NY, USA, 2007. ACM.

[40] L. Candela, F. Akal, H. Avancini, D. Castelli, L. Fusco, V. Guidetti, C. Langguth, A. Manzi, P. Pagano, H. Schuldt, M. Simi, M. Springmann, and L. Voicu. DILIGENT: integrating digital library and Grid technologies for a new Earth observation research infrastructure. *International Journal of Digital Libraries*, 7(1):59–80, 2007.

[41] L. Candela, D. Castelli, C. Langguth, P. Pagano, H. Schuldt, M. Simi, and L. Voicu. On-Demand Service Deployment and Process Support in e-Science DLs: the DILIGENT Experience. In *ECDL Workshop on Digital Library goes eScience (DLSci06)*, September 2006.

[42] R. L. Carter and M. E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical report, Computer Science Department, Boston University, 1996.

[43] D. Castelli. e-Infrastructures designed for demanding science. *eStrategies Europe, "Europe's new flagship for innovation"*, 2(4), November 2008.

[44] D. Castelli and J. M. J. D4Science – Deploying Virtual Research Environments. In *ERCIM News 74 Special theme: Supercomputing at Work*, pages 8–9, July 2008.

[45] S. Ceri and G. Pelagatti. *Distributed databases principles and systems*. McGraw-Hill, Inc., New York, NY, USA, 1984.

[46] CERN. LHC Computing Centres Join Forces for Global Grid Challenge. CERN Press Release, 2005. `http://press.web.cern.ch/press/PressReleases/Releases2005`.

[47] A. Chan, S. Fox, W.-T. K. Lin, A. Nori, and D. R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 184–191, New York, NY, USA, 1982. ACM.

[48] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, 11(2):205–212, 1985.

[49] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[50] A. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf. Performance and Scalability of a Replica Location Service. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 181–192, June 2004.

[51] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe. Wide area data replication for scientific collaborations. *Proceedings of the 6th International Workshop on Grid Computing*, November 2005.

[52] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):582–592, 1992.

[53] The Chimera: A Virtual Data System. http://www.griphyn.org/chimera/.

[54] J. Cho and H. Garcia-Molina. Synchronizing a database to Improve Freshness. In *SIGMOD*, pages 117–128, 2000.

[55] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred Updates and Data Placement in Distributed Databases. In *ICDE*, pages 469–476, 1996.

[56] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM Comput. Commun. Rev.*, pages 177–190, 2002.

[57] D. L. Collins, J. Montagnat, A. P. Zijdenbos, A. C. Evans, and D. L. Arnold. Automated estimation of brain volume in multiple sclerosis with biccr. In *IPMI '01: Proceedings of the 17th International Conference on Information Processing in Medical Imaging*, pages 141–147, London, UK, 2001. Springer-Verlag.

[58] G. Comi, M. Philippi, V. Martinelli, G. Sirabian, A. Visciani, A. Campi, S. Mammi, M. Rovari, and M. Canal. Brain magnetic resonance imaging correlates of cognitive impairment in multiple sclerosis. *Journal of the Neurological Science*, 115:66–73, 1993.

[59] B. Cooper, E. Baldeschwieler, et al. Building a Cloud for Yahoo! *IEEE DE Bulletin*, 32(1):36–43, 2009.

[60] B. Cooper, E. Baldeschwieler, R. Fonseca, J. Kistler, P. Narayan, C. Neerdaels, T. Negrin, R. Ramakrishnan, A. Silberstein, U. Srivastava, and R. Stata. Building a Cloud for Yahoo! *IEEE Data Engineering Bulletin*, 32(1):36–43, 2009.

[61] B. Cooper, R. Ramakrishnan, et al. PNUTS: Yahoo!'s Hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, 2008.

[62] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, 2008.

[63] DAIS-WG. https://forge.gridforum.org/projects/dais-wg.

[64] G. Dasgupta, K. Dasgupta, A. Purohit, and B. Viswanathan. QoS-GRAF: A Framework for QoS based Grid Resource Allocation with Failure provisioning. *14th IEEE International Workshop on Quality of Service*, pages 281–283, June 2006.

[65] A. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Y. Yao. The Cougar Project: a work-in-progress report. In *SIGMOD Record*, volume 32, pages 53–59, 2003.

[66] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM.

[67] Apache derby. http://db.apache.org/derby/.

[68] M. Deris, J. Abawajy, and H. Suzuri. An efficient replicated data access approach for large-scale distributed systems. *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 588–594, April 2004.

[69] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-Driven Data Acquisition in Sensor Networks. In *VLDB*, pages 588–599, 2004.

[70] EDG: The European DataGrid Project. http://eu-datagrid.web.cern.ch/eu-datagrid/.

[71] EGEE: The Enabling Grids for E-sciencE Project. http://www.eu-egee.org/.

[72] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: memory-aware load balancing and update filtering in replicated databases. *SIGOPS Oper. Syst. Rev.*, 41(3):399–412, 2007.

[73] ESA - The European Space Agency. http://www.esa.int/.

[74] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[75] FAST, Demystifying Search. http://www.fast.no/l3a.aspx?m=43.

[76] A. Fekete. Allocating isolation levels to transactions. In *PODS*, pages 206–215, New York, USA, 2005. ACM.

[77] I. Foster. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Global Grid Forum*, 2002.

[78] I. Foster, K. Czajkowski, D. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. Modeling and Managing State in Distributed Systems: The Role of OGSI and WSRF. *Proceedings of the IEEE*, 93(3):604–612, March 2005.

[79] I. Foster, D. Gannon, H. Kishimoto, and J. J. V. Reich. Open Grid Services Architecture Use Case. Technical report, Open Grid Services Architecture, 2004.

[80] I. Foster and C. Kesselman. The Globus project: a status report. *Future Generation Computer Systems*, 15(5–6):607–621, 1999.

[81] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2nd edition, 2003.

[82] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15:2001, 2001.

[83] R. Gallersdörfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 445–456, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[84] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.

[85] S. Genaud and C. Rattanapoka. P2P-MPI: A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs on Grids. *Journal of Grid Computing*, 5(1), March 2007.

[86] IBM alphaWorks, Grid File Replication Manager. http://www.alphaworks.ibm.com/tech/gfrm.

[87] The Globus Alliance. http://www.globus.org/.

[88] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive Replication in Peer-to-Peer Systems. In *ICDCS*, pages 360–369, 2003.

[89] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

[90] J. Gray. The Revolution in Database Architecture. In *Extended abstract of keynote talk at ACM SIGMOD 2004*, June 2004.

[91] J. Gray et al. Scientific Data Management in the Coming Decade. In *MSR-TR-2005-10*, January 2005.

[92] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *International Conference on Management of Data*, pages 173–182, 1996.

[93] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31:2006, 2003.

[94] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

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

[96] J. Gray, A. Szalay, A. Thakar, P. Kunszt, C. Stoughton, D. Slutz, and J. van den Berg. The Sloan Digital Sky Survey Science Archive: Migrating a Multi-Terabyte Astronomical Archive from Object to Relational DBMS. In *Distributed Data Structures 4: Records of the 4th International Meeting*, pages 189–210, March 2002.

[97] GridFTP, Universal Data Transfer for the Grid. http://www.globus.org/toolkit/docs/2.4/datagrid/ deliverables/C2WPdraft3.pdf.

[98] The Globus Toolkit. http://www.globus.org/toolkit/.

[99] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15:287–317, 1983.

[100] K. Haller. *Transaktionsverwaltung für Dienstorientierte Architekturen - Ein Peer-To-Peer-Basierter Ansatz*. Diss. ETH No. 15974, ETH Zurich, 2005.

[101] K. Haller, H. Schuldt, and C. Türker. Decentralized Coordination of Transactional Processes in Peer-to-Peer Environments. In *Proc. CIKM'05*, pages 28–35, Bremen, Germany, 2005.

[102] R. Harris and N. Olby. Archives for Earth observation data. *Space Policy*, 16:223–227, 2007.

[103] G. Heber and J. Gray. Supporting Finite Element Analysis with a Relational Database Backend Part I: There is Life beyond Files. In *MSR-TR-2005-49*, April 2005.

[104] J. Hellerstein, S. Madden, M. Franklin, , and W. Hong. TinyDB: An Acqusitional Query Processing System for Sensor Networks. *ACM Trans. on Database Systems*, 2005.

[105] W. Hoschek, J. Jaen-Martinez, A. Samar, and H. Stockinger. Data Management in an International Data Grid Project. *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (Lecture Notes in Computer Science)*, 1971:77–90, 2000.

[106] IBM WebSphere Process Choreographer. http://www-106.ibm.com/developerworks/zones/was/wpc.html.

[107] How Much Information? – University of California at Berkeley Study. http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/printable_report.pdf .

[108] R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Are Quorums an Alternative For Data Replication? *ACM Transactions on Database Systems*, 28:2003, 2003.

[109] R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Enterprise Grids: Challenges Ahead. *Grid Computing*, 5(3):283–294, 2007.

[110] B. Kemme and G. Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems*, 25:2000, 2000.

[111] B. Kemme, G. Alonso, F. Pedone, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. *Distributed Computing Systems, International Conference on*, 0:0424, 1999.

[112] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[113] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. Technical report, Austin, TX, USA, 1990.

[114] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[115] C. Lac and S. Ramanathan. A Resilient Telco Grid Middleware. *Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 306–311, June 2006.

[116] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 2001.

[117] M. Lei, S. Vrbsky, and Q. Zijie. Online Grid Replication Optimizers to Improve System Reliability. *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[118] The Laser Interferometer Gravitational Wave Observatory. http://www.ligo.caltech.edu/.

[119] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. In *SIGMOD*, pages 419–430, 2005.

[120] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Enhancing Edge Computing With Database Replication. In *IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, China, 2007.

[121] P. Liu and J.-J. Wu. Optimal Replica Placement Strategy for Hierarchical Data Grid Systems. *Proceedings of the 16th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, 0:417–420, 2006.

[122] N. A. Losseff, L. Wang, H. M. Lai, D. S. Yoo, A. Visciani, M. L. Gawne-Caine, W. I. McDonald, D. H. Miller, and A. J. Thomson. Progressive cerebral atrophy in multiple sclerosis, a serial MRI study. *Brain*, 119(6):2009–2019, 1996.

[123] The Globus Replica Location Service. http://www.isi.edu/annc/research/RLSsummary.pdf.

[124] M. Manohar, A. Chervenak, B. Clifford, and C. Kesselman. A Replica Location Grid Service Implementation. In *Data Area Workshop, Global grid Forum*, 2004.

[125] R. W. Moore, A. Rajasekar, and M. Wan. Data Grids, digital libraries and persistent archives: An integrated approach to publishing, sharing and archiving data. In *Proceedings of the IEEE (Special Issue on Grid Computing)*, 2005.

[126] J. E. B. Moss. *Nested transactions: an approach to reliable distributed computing*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[127] J. E. B. Moss. Nested transactions: an introduction. pages 395–425, 1987.

[128] M. Nieto-Santisteban, A. Szalay, A. Thakar, W. O'Mullane, J. Gray, and J. Annis. When Database Systems Meet The Grid. In *ACM CIDR*, January 2005.

[129] R. Obermack. Distributed deadlock detection algorithms. *ACM Transactions on Database Systems*, 7(2):187–208, 1982.

[130] OGSA-DAI. http://www.ogsa-dai.org.uk/.

[131] OGSA-DQP. http://www.ogsa-dai.org.uk/dqp.

[132] Data Grids and Service-Oriented Architecture, an Oracle White Paper, Oracle Corporation. http://www.oracle.com/technologies/grid/index.html.

[133] M. T. Özsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, USA, 1999.

[134] E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 126–137, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[135] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed Parallel Databases*, 9(3):237–267, 2001.

[136] E. Pacitti, M. T. Özsu, and C. Coulon. Preventive Multi-Master Replication in a Cluster of Autonomous Databases. In *In Euro-Par*, pages 318–327, 2003.

[137] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *The VLDB Journal*, 8(3-4):305–318, 2000.

[138] E. Pacitti, E. Simon, and R. Melo. Improving Data Freshness in Lazy Master Schemes. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 164, Washington, DC, USA, 1998. IEEE Computer Society.

[139] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, Inc., New York, NY, USA, 1986.

[140] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[141] J.-F. Pâris and D. D. E. Long. Efficient dynamic voting algorithms. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 268–275, Washington, DC, USA, 1988. IEEE Computer Society.

[142] N. Paton, M. Aragão, K. Lee, A. Fernandes, and R. Sakellariou. Optimizing Utility in Cloud Computing through Autonomic Workload Execution. *IEEE DE Bulletin*, 32(1):51–58, 2009.

[143] N. Paton, M. Atkinson, V. Dialani, D. Pearson, T. Storey, and P. Watson. Databases Access and Integration Services on the Grid. Technical report, UK e-Science Programme, National e-Science Centre, 2003.

[144] D. Pearson. Data Requirements for the Grid. Technical report, Technical Report, 2002.

[145] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach.

[146] Pegasus. http://pegasus.isi.edu/.

[147] Z. Qi, X. Xie, B. Zhang, and J. You. Integrating X/Open DTP into Grid Services for Grid Transaction Processing. In *The 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04)*, pages 128–134, 2004.

[148] Z. Qiwei, Y. Jiangming, G. Ning, Z. Yuwei, D. Zhigang, and Z. Shaohua. Dynamic Replica Location Service Supporting Data Grid Systems. *Proceedings of the 6th IEEE International Conference on Computer and Information Technology (CIT'06)*, pages 61–61, September 2006.

[149] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. In *EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology*, pages 207–222, London, UK, 1996. Springer-Verlag.

[150] B. Radic, V. Kajic, and E. Imamagic. Optimization of data transfer for grid using gridftp. In *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on*, pages 709–715, June 2007.

[151] R. Ramakrishnan. Data Management in the Cloud. In *Proceedings of the 25th International Conference on Data Engineering (ICDE 2009)*, page 5, Shanghai, China, March-April 2009.

[152] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):997–1007, 1995.

[153] K. A. Rathore, S. K. Madria, and T. Hara. Adaptive searching and replication of images in mobile hierarchical peer-to-peer networks. *Data Knowl. Eng.*, 63(3):894–918, 2007.

[154] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Cambridge, MA, USA, 1978.

[155] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, 1983.

[156] M. Ripeanu and I. Foster. A Decentralized, Adaptive Replica Location Mechanism. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, 0:24–32, 2002.

[157] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. FAS: a freshness-sensitive coordination middleware for a cluster of OLAP components. In *VLDB*, pages 754–765, 2002.

[158] U. Röhm and S. Schmidt. Freshness-Aware Caching in a Cluster of J2EE Application Servers. In *WISE*, pages 74–86, 2007.

[159] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *Computer*, 24(12):46–53, 1991.

[160] H.-J. Schek, G. Weikum, and H. Ye. Towards a unified theory of concurrency control and recovery. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 300–311, New York, NY, USA, 1993. ACM.

[161] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Commun. ACM*, 39(4):84–87, 1996.

[162] G. Schlageter. Optimistic methods for concurrency control in distributed database systems. In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 125–130. VLDB Endowment, 1981.

[163] C. Schuler, R. Weber, H. Schuldt, and H.-J. Schek. Scalable peer-to-peer process management - the osiris approach. *Web Services, IEEE International Conference on*, 0:26, 2004.

[164] L. Seligman and L. Kerschberg. A mediator for approximate consistency: Supporting "goodenough" materialized views. *J. Intell. Inf. Syst.*, 8(3):203–225, 1997.

[165] G. Sergey, E. Berman, C. H. Huang, S. Kent, H. Newberg, T. Nicinski, D. Petravick, C. Stoughton, and R. Lupton. Shiva: an astronomical data analysis framework. In *ASP Conf. Ser. 101: Astronomical Data Analysis Software and Systems*, 1996.

[166] A. Sheth and M. Rusinkiewicz. Management of interdependent data: specifying dependency and consistency requirements. In *Management of Replicated Data, 1990. Proceedings., Workshop on the*, pages 133–136, Nov 1990.

[167] F. Simeoni, D. Castelli, P. Pagano, M. Simi, and R. Connor. Application-level Research e-Infrastructures: the gCube Approach. In *UK e-Science All Hands Meeting 2008*, September 2008.

[168] S. H. Son and S. Kouloumbis. A token-based synchronization scheme using epsilon-serializability and its performance for real-time distributed database systems. In *Proceedings of the 3rd International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 235–242. World Scientific Press, 1993.

[169] SRB: The Storage Resource Broker. http://www.sdsc.edu/srb/.

[170] I. Stanoi, D. Agrawal, and A. E. Abbadi. Using broadcast primitives in replicated databases. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, page 283, New York, NY, USA, 1997. ACM.

[171] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and Object Replication in Data Grids. *Proceedings of 10th IEEE Symposium on High Performance and Distributed Computing*, 2001.

[172] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Trans. Softw. Eng.*, 5(3):188–194, 1979.

[173] S. Takizawa, Y. Takamiya, H. Nakada, and S. Matsuoka. A Scalable Multi-Replication Framework for Data Grid. *Proceedings of the 2005 Symposium on Applications and the Internet Workshops (SAINT-W'05)*, pages 310–315, January 2005.

[174] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. In *CC-GRID*, pages 102–110, 2002.

[175] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[176] S. Tuecke et al. Open Grid Services Infrastructure (OGSI), Version 1.0. *Proposed Recommendation, Open Grid Forum*, 2003.

[177] C. Türker, K. Haller, C. Schuler, and H.-J. Schek. How can we support Grid Transactions? Towards Peer-to-Peer Transaction Processing. In *CIDR*, pages 174–185, 2005.

[178] L. Valcarenghi and P. Castoldi. QoS-Aware Connection Resilience for Network-Aware Grid Computing Fault Tolerance. *International Conference on Transparent Optical Networks*, 1:417–422, 2005.

[179] D. Verma, S. Sahu, S. Calo, A. Shaikh, I. Chang, and A. Acharya. SRIRAM: A scalable resilient autonomic mesh. Technical report, IBM Systems Journal, 2003.

[180] R. Vingralek, Y. Breitbart, and G. Weikum. Snowball: Scalable Storage on Networks of Workstations with Balanced Load. *Distributed and Parallel Databases*, 6(2):117–156, 1998.

[181] R. Vingralek, H. Hasse-Ye, Y. Breitbart, and H.-J. Schek. Unifying concurrency control and recovery of transactions with semantically rich operations. *Theoretical Computer Science*, 190(2), 1998.

[182] R. Vingralek, M. Sayal, P. Scheuermann, and Y. Breitbart. Web++: A system for fast and reliable web service. In *USENIX Annual Technical Conference*, pages 6–11, 1999.

[183] L. Voicu and H. Schuldt. The Re:GRIDiT Protocol: Correctness of Distributed Concurrency Control in the Data Grid in the Presence of Replication. Technical report, University of Basel, Department of Computer Science, 2008.

[184] L. C. Voicu and H. Schuldt. How Replicated Data Management in the Cloud can benefit from a Data Grid Protocol – the Re:GRIDiT Approach. In *Proceedings of the 1st International Workshop on Cloud Data Management (CloudDB'09)*, November 2009.

[185] L. C. Voicu and H. Schuldt. Load-aware Dynamic Replication Management in a Data Grid. In *Proceedings of the 17th International Conference on Cooperative Information Systems (CoopIS'09)*, November 2009.

[186] L. C. Voicu, H. Schuldt, F. Akal, Y. Breitbart, and H.-J. Schek. Re:GRIDiT – Coordinating Distributed Update Transactions on Replicated Data in the Grid. In *Proceedings of the 10th IEEE/ACM on Grid Computing (Grid'09)*, October 2009.

[187] L. C. Voicu, H. Schuldt, Y. Breitbart, and H.-J. Schek. Replicated Data Management in the Grid: the Re:GRIDiT Approach. In *ACM Workshop on Data Grids for eScience (DaGreS'09)*, May 2009.

[188] W. E. Weihl. Distributed version management for read-only actions. *IEEE Trans. Softw. Eng.*, 13(1):55–64, 1987.

[189] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Trans. Database Syst.*, 16(1):132–180, 1991.

[190] G. Weikum and H.-J. Schek. Architectural issues of transaction management in multi-layered systems. In *VLDB '84: Proceedings of the 10th International Conference on Very Large Data Bases*, pages 454–465, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[191] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. *Database transaction mls for advanced applications*, pages 515–553, 1992.

[192] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, San Francisco, California, 2001.

[193] J. Weissman and B. Lee. The Virtual Service Grid: an Architecture for Delivering High-End Network Services. *Concurrency And Computation*, 14:287–319, 2002.

[194] OASIS Web Services Composite Application Framework. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf.

[195] The Web Services Transactions Specifications. www.ibm.com/developerworks/library/specification/ws-tx/.

[196] S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. In *ICDE*, pages 422–433, 2005.

[197] Distributed Transaction Processing: The XA Specification. www.opengroup.org/onlinepubs/009680699/toc.pdf.

[198] J. Xu, P. Townend, N. Looker, and P. Groth. FT-Grid: a System for Achieving Fault Tolerance in Grids. *Concurrency and Computation: Practice and Experience*, 20(3):297–309, 2008.

[199] X. Zhang, F. Junqueira, M. Hiltunen, K. Marzullo, and R. Schlichting. Replicating Nondeterministic Services on Grid Environments. *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, pages 105–116, 2006.

[200] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. Schlichting. Fault-tolerant grid services using primary-backup: feasibility and performance. *IEEE International Conference on Cluster Computing*, pages 105–114, September 2004.

Curriculum Vitae

Laura Cristiana Voicu

May 24, 1979	Born in Craiova, Romania Daughter of Viorica and Dorel Voicu
1986–1990	Primary school, Piteşti, Romania
1990–1994	Secondary school, Gymnasium, Piteşti, Romania
1994–1998	High-school, College, Piteşti, Romania
1998–2003	Study of Computer Science Engineering, Universitatea Politehnica Bucureşti, Romania
2002–2004	Study of Physics, Universität Siegen, Germany
2003	Dipl. Engineer in Computer Science Engineering, Universitatea Politehnica Bucureşti, Romania
2004	M.Sc. in Physics, Universität Siegen, Germany
2004–2005	Research assistant in the group of Prof. D. Cowen, Physics Department, Pennsylvania State University, USA.
2005–2006	Research assistant in the group of Prof. H.-J. Schek, Institute for Information Systems, ETH Zurich, Switzerland and Institute for Information Systems, UMIT Hall in Tyrol, Austria.
2006–2009	Research assistant in the group of Prof. Heiko Schuldt, Database and Information Systems, Universität Basel, Switzerland.