The Re:GRIDiT Protocol: Correctness of Distributed Concurrency Control in the Data Grid in the Presence of Replication

Laura Voicu and Heiko Schuldt

Technical Report CS-2008-002
University of Basel
Email: {laura.voicu, heiko.schuldt}@unibas.ch

Abstract

In recent years, advances in high performance distributed computing have brought forward novel types of applications which access and share resources across heterogeneous, distributed environments. However, at the same time, it has to be noted that current approaches to ensure the correctness of the execution of applications in such distributed environments in the presence of concurrent updates do not scale and are not able to cope with the dynamic nature of these environments, particularly when the replication of huge amounts of data has to be considered. Due to the particular characteristics of such infrastructures (such as data Grids or peer-to-peer networks), especially due to the absence of a global coordinator, new protocols for the synchronization of updates and their subsequent propagation are urgently needed. Despite the considerable work which has been done in the context of distributed transaction management and replication management, there is no protocol which can be seamlessly applied to such environments without impacting correctness and/or overall performance. In this paper we propose Re:GRIDiT, a novel and flexible distributed transactional model, that provides seamless support of replicated data and provably correct transactional execution guarantees without any global component.

Keywords:
Distributed transaction management, Data Grid, Replication, Peer-to-peer transaction management, Correctness.

Computer Science Department, Databases and Information Systems Group, Bernoullistrasse 16, CH – 4056 Basel Switzerland. Phone: +41 61 267 05 32 URL: http://dbis.cs.unibas.ch/

1 Introduction

In recent years, the advances in high performance distributed computing have brought forward new types of applications. They address solving larger and more complex problems, realizing more reliable and efficient applications that deal with highly distributed and heterogeneous data and providing distributed, heterogeneous, and dynamic resources which span the boundaries of organizations.

A lot of effort has been put into consolidating open distributed environments (Grid computing environments, cloud computing environments, or peer-to-peer systems being the most noteworthy examples). Still, there is a significant gap between what is required by newly emerging domains, such as eHealth and eScience, and what is currently provided by available technologies.

In the field of eHealth for instance, 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 mutability of data – ranging from read-only data (e.g., medical images) to data which is frequently updated, annotated, and/or appended (e.g., medical records)
- 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: (i) Access efficiency (moving data near processing), (ii) Load balancing (distributing access load), (iii) Security (data protection, moving processing near data if data confidentiality is an issue), (iv) Availability (off-line access), (v) Reliability (disaster recovery, avoiding single point of failure).

Recent advances in Grid computing tackle some of these problems by virtualizing the resources (data, instruments, computing nodes, tools, and users) and making them transparently available. But whereas some key issues can be solved using today's Grid technologies, in other aspects Grid technologies are still in their youth and often propose only very generic services.

1.1 A Sample eHealth Application

Taking a closer look at application scenarios coming from the eHealth domain, we notice more and more that 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. New technologies that face the raising challenges of computational medicine, 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 are more and more needed. 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 [1, 2]. 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 [3].

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 healthcare 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 but 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 in what concerns 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 network to be made available to hundreds of scientists working in the same field and sharing the same data.

1.2 Grid Data Management

The original vision of Grid computing was characterized by the need to link together computing resources, but, as it can be seen from the examples above, newly emerging fields impose different requirements. With increased computing resources comes also the need for efficient data management protocols. Running a large study with Grid resources will necessarily lead to the production of huge amounts of data, too many to be able to manage using conventional approaches. The data management challenge is further compounded by the fact that simulations performed on distributed computing environments will lead to data being stored in several locations and a global component is no longer a feasible approach (a central coordinator will soon become a hot-spot and a single point of failure). Availability, dependability and scalability are still 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. We believe Grids will eventually become a staple of mainstream computing, and an efficient and flexible data replication management protocol

will enhance the tremendous potential in Grid technologies to face today's medical application challenges. For this purpose, we see the Grid as a network of sites, where efficient replication mechanisms maintain numerous replicas consistent and keep the intricacies of data movement, synchronization, error handling, load balancing, etc. transparent to the user. We consider the sites where the medical images are inserted or where scientists update their patient reports to be updateable. In order to improve availability copies of data are maintained at read-only sites.

In order to purse this vision and as a first step to solve the above application scenario, we have developed Re:GRIDiT, a protocol which manages the synchronization of updates to several replicas in a completely distributed way. Globally correct execution is provided by communication between transactions and sites. The contribution of our work is the following: we propose the first protocol that meets the challenges of replication management and that provides provably correct transactional execution guarantees without any global component.

The remainder of this technical report is organized as follows. Section 2 discusses related work. In Section 3, we explain our system model. Section 4 gives details of the protocol we propose, a proof of correctness of the protocol and protocol optimizations. Finally, Section 5 concludes.

2 Related Work

We are faced nowadays with the need for more and more complex computations on huge amounts data that take place in open distributed infrastructures. There exists naturally the need to ensure a correct execution and to guarantee the success of such computations, which can be achieved by enforcing transactional guarantees on their execution. However, it is commonly agreed that the rigid database concepts of atomicity and isolation are no longer suited for composite and complex applications. Rather, advanced and extended transaction models that seamlessly allow compensations and alternatives need further consideration. The proliferation of web services and the interest towards the use of transactions in service oriented computing have gained attention in the past years, resulting in several specifications (for example [4]), some of them supported by the computer industry and which try to offer an alternative to the traditional concepts of database transactions. Since these models are based on the principle that globally correct execution of transactional processes is ensured by the existence of a global coordinator, they do not scale any longer in the emerging large-scale distributed environments. A single coordinator in a distributed environment can soon become a hot-spot and a potential single point of failure – if the coordinator becomes unavailable then no transaction can commit.

A decentralized serialization graph testing protocol is proposed in [5, 6] that ensures concurrency control and recovery in peer-to-peer environments. The uniqueness of the proposed protocol is that it ensures global correctness without relying on a global serialization graph. Essentially, each transaction is equipped with partial knowledge that allows the transactions to coordinate. Globally correct execution is achieved by communication among dependent transactions and the peers they have accessed. In case of failures, a combination of partial backward and forward recovery is applied. However, the proposed protocol does not take into account replication. Yet replication is an important advantage of large-scale environments since it allows to provide and guarantee a high degree of availability of data. GridTP is another approach to transaction management in a service-oriented Grid [7] which is based on the Open Grid Services Architecture (OGSA) platform and the X/Open DTP model. This means that a two-phase commit protocol is

used to atomically commit distributed Grid transactions, thus relying again on the existence of a global coordinator.

In terms of data management, the Grid allows keeping a large number of replicas of data objects, maybe even with different levels of freshness, to ensure a high degree of availability, reliability and performance. Replication management in the Grid has to deal with a potentially large number of updateable replicas per data object. Due to the particular characteristics of the Grid, especially due to the absence of a global coordinator, this requires new protocols for the synchronization of updates and their subsequent propagation. 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. To develop such management schemes one needs to resolve data availability and data consistency issues for the data Grid environment. One of the constraints current data Grids are suffering from is that Grid files are mainly considered as read-only [8, 9]. Furthermore, for most of the available solutions, replication is managed manually and based on single files as the replication granularity [10]. There are also more comprehensive solutions as proposed in [11]. This solution, however, may suffer from consistency problems in case of multiple catalogs and while accessing more than one replica.

Several replication solutions, on the other hand, for databases and database clusters already exist in the literature that may be partially applicable to distributed environments (such as [12], [13] or [14]). [14] is a recent protocol for database replication that supports freshness and lazy update propagation for many read-only nodes. 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. In fact, 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 such environments. In [13], several updateable replicas are supported but a single, global replication graph is required. Similarly, [12] works under the assumption that one replica copy is designated as the primary copy, stored at a master node, and update transactions are only allowed on that replica.

Chord [15] or CAN [16] are two well-established, hash-based peer-to-peer systems. Many approaches for peer-to-peer replication have been designed based on these prototypical systems (for example [17, 18, 19]). Unlike many other approaches where the goal is to trade consistency for performance, [18] focuses on returning the current (most recent) replica, while ensuring consistency guarantees. Other approaches, like [19, 17] do not deal with data consistency issues, restricting their focus on read-only data (new copies of data can only be created or deleted, but never updated).

Thus, none of the existing replication protocols can be fully decentralized which is an important requirement in these large-scale distributed infrastructures or offer the degree of flexibility need by applications coming from the eHealth or eScience domain.

Our approach extends and generalizes the approaches presented in [5]. In [20] we sketch how we can adapt these protocols to the Grid. In this paper we define Re:GRIDiT, a transaction protocol for replicated Grid data, that generalizes the approach proposed in [5] by extending it to support replication.

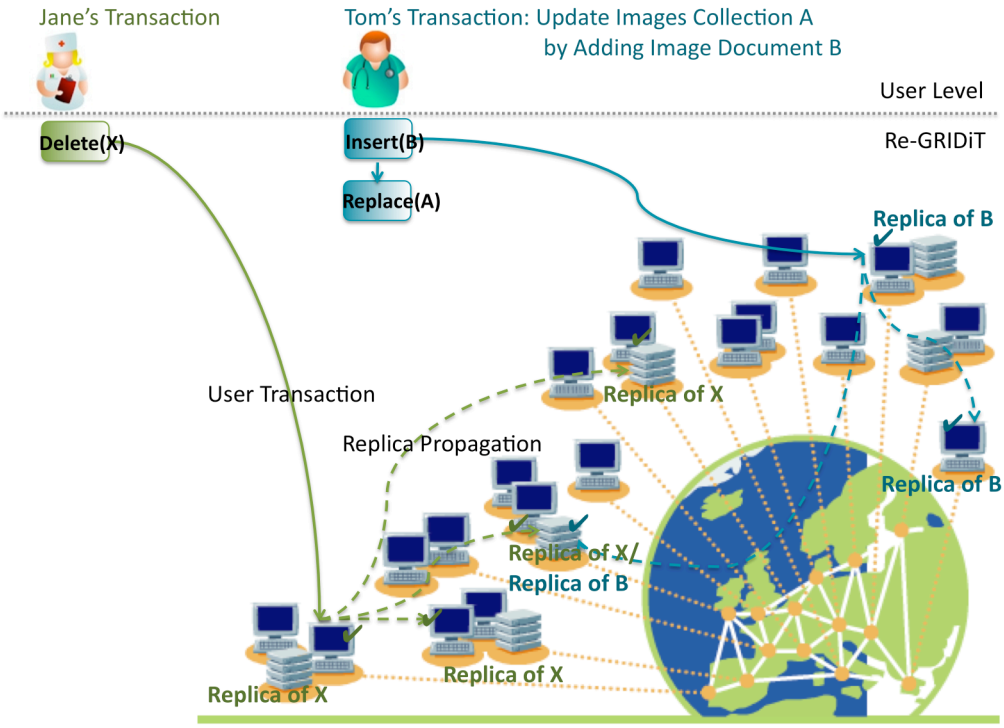

Figure 1: Re:GRIDiT System Overview.

3 Re:GRIDiT – System Model

Re:GRIDiT (Replication Management in Data GRid Infrastructures using Distributed Transactions) was developed as a response to the need of a protocol that meets the challenges of replication management in the 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. Our protocol is characterized by the following contributions:

- Distributed concurrency control: user operations span several sites, i.e., require support for distributed transaction management.
- Replication management: we divide the sites between updateable (where data objects can be updated) and read-only sites (only read access to data is allowed). We assume replication among updateable sites as well as replication between updateable and read-only sites. Our assumptions lead to the necessity to correctly detect and handle remote conflicts transparently. Once the update sites are synchronized among themselves, any read-only replica in the network can be updated using a mechanism similar to the one proposed in [14]. For this purpose we concentrate our efforts on the coordination of updates among the update sites.
- No global coordinator: our approach enforces globally serializable schedules in a completely distributed way without relying on a central coordinator that has complete global knowledge. The proposed approach treats all sites of the Grid in a uniform way, i.e., it does not assume any dedicated sites.

Figure 1 offers a general overview of the Re:GRIDiT system. Our proposal of a completely distributed transaction protocol relies on several model assumptions, which we present in the following paragraphs (illustrated in Figure 2).

We assume a distributed 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.

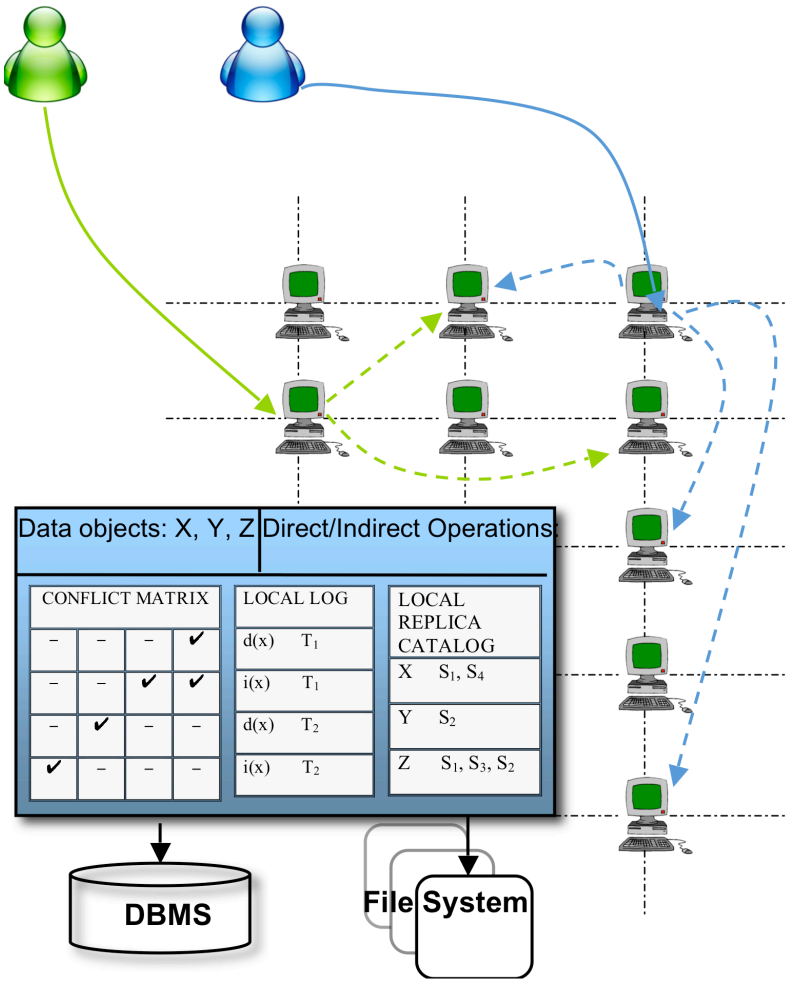

Figure 2: System Model (detailed view).

The sites hold data objects, which are replicated in the network on several other sites, in order to ensure a high degree of availability. We assume in our approach that one physical object can reside only on one site[1], but can be replicated on several others. We assume furthermore that every site provides a basic infrastructure consisting of (any) relational database and a file management system, where the data objects can be stored. Each site offers a set of semantically rich operations acting on the data objects, and we rely in our protocol on the assumption that operation invocations on objects stored in the database are executed as local database transactions, and operation invocations on objects stored on the local file system rely on file management functionality, such as GridFTP [21]. Without entering into the details of these operations (our protocol is independent on the choice of the operations supported, in fact, any set of semantically rich operations can be used for this purpose), we rely on the existence of semantically inverse operations so that the effect of an operation invocation can be semantically compensated in case of a failure. Moreover, we assume that there exists a local replica catalog (as proposed in [22]) through which the configuration of replicas in the network in known, that is, any site in the network is aware of which other sites are holding replicas of the same data objects. We assume for the sake of simplicity that this configuration is static, however we aim at supporting dynamic replica placement and deployment. Possible conflicts between operations are are reflected in conflict matrices (as described in Figure 3), available on all sites. By using these conflict matrices, the sites can automatically detect conflicts. All conflicts are stored in a local log on the sites.

[1]Nevertheless, one site can be composed of a database cluster

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

Figure 3: Conflict Matrix.

This basic functionality, which can be considered an abstraction from basic Grid storage functionality, is assumed to be present on each site. Re:GRIDiT is envisioned to sit on top of this layer and is envisioned to be present on each site as part of the Grid infrastructure, providing transparent support for replication and distributed transaction management.

In our model we distinguish between two types of update transactions: *direct* and *indirect update transactions*. Direct update transactions are initiated by the user and the middleware directs them to the appropriate update site by using a distributed replica catalog, according to which the sites holding data relevant to the transaction are chosen. Indirect update transactions are transparent to the user, and are initiated by the sites, in order to support replication, as follows: when a direct update transaction T invokes an operation on a data object d on site s, the site will according to our protocol take care that the operation invocation is replicated on all the sites in the network that hold a replica of the same data object d. S will trigger an indirect update transaction on behalf of T that will update d on every site, within the boundaries of the initiating direct update transaction T (similar to a nested transaction). Consequently, the operations available on each site can be divided into: *direct operations*, which belong to direct update transactions (as invoked by the user), and *indirect operations*, which belong to indirect update transactions (in order to support replication).

Figure 3 provides a sample conflict matrix and also exemplifies the distinction between direct and indirect operations. The set of operations comprises three direct operations, namely `Insert`, `Replace`, and `Read`. The indirect operations which are transparent to the user since they are automatically added to a user's transaction are `Copy` and `Remove`. A `Copy` operation is used, for instance, to create new replicas after a new data object has been created using the direct `Insert` operation. A direct `Replace` operation will be propagated by issuing a `Remove` operation followed by a `Copy` operation at each site where a replica resides.

4 Re:GRIDiT – Protocol and Correctness

Re:GRIDiT is based on optimistic transaction management techniques and relies on communication between transactions and sites to ensure global correctness (extending the DSGT protocol presented in [5]). Nevertheless, Re:GRIDiT is not a simple generalization of an existing protocol; its novel features, which make it uniquely flexible, are *transparent support for data replication at Grid scale* and *provably correct transactional guarantees without a global coordinator*. Although Re:GRIDiT has originally been designed for the support of distributed transactions in a Data Grid in the presence of replication, it can be seamlessly applied to other environments which feature the same or similar characteristics, especially the lack of a global coordinator. This includes data and replication management in peer-to-peer overlay networks and replicated data management in cloud computing.

The Re:GRIDiT protocol aims at providing reliable support for replication management and builds upon the following observations:

- Dependencies between transactions are managed by the transactions themselves.

- Global correctness can be achieved even in the absence of a global coordinator and with incomplete knowledge by communication between the transactions and the sites.
- Replication is handled by the sites, completely transparent to the transactions and the users.

4.1 Overview of the Re:GRIDiT Protocol

A transaction invokes operations on data objects optimistically, without requesting any locks. The sites where the transactions have invoked operations reply with a list of local conflicts, based on the local conflict matrix. Since we assume a replicated system model, each operation on a data object on a site will be propagated to all the sites in the system that contain a replica of that particular data object. In our protocol we free the original direct update transaction from these indirect update operations, by delegating the invocation of the latter 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. We assume a FIFO order of the triggered indirect operations and that the communication between the sites is reliable. For the originating transaction it is not important to know immediately if one of the indirect update 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 update operations, and update their serialization graphs accordingly.

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

The Re:GRIDiT protocol requires, as mentioned before, communication between the transactions and the sites for the execution of direct operations and communication between sites for the execution of indirect operations. Therefore the Re:GRIDiT algorithm is decoupled into two parts, one running on each site, the other part for each transaction. The part of the protocol that runs for each transaction consists of three phases (execution, validation, and commit).

1. *The execution phase:* The transaction invokes its operations according by using 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 erialization 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 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.

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, we also need 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 remote 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. However, prior to commit a validation step is performed in order to establish if the transaction has been executed correctly or if it has generated any conflicts on remote sites. The transaction will determine on its own if it is allowed to commit.

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 [23], 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. (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 [24]), it aborts.

4.2 Re:GRIDiT in Detail

The Re:GRIDiT protocol is implemented according to the following algorithms:

4.2.1. Transaction Protocol

```
1 // sequence of direct operations to be invoked by T;
2 O*_T := [o_1, ..., o_n]
3 // sites on which T invoked operations;
4 S*_T := {}
5 // local copy of the serialization graph;
6 SG_T := {}
7
8 Main Thread:
9 // 1. Execution Phase:
10 for (o_i ∈ O*_T) {
11     //invoke operations and update the serialization graph
12     invoke o_i to an appropriate site s and add s to S*_T;
13     wait for reply from s;
14     // site will eventually send return values of the
15     // operation and a list of local conflicts
16     // and update the graph with new conflicts
17     update SG_T based on reply information;
18     if (new dependencies emerged){
19         //propagate changes to pre-ordered transactions
20         propagate SG_T to pre-ordered transactions;
21     }
22 }
```

```
// 2. Validation Phase:
for (s ∈ S*_T){
    request update from s;
    wait for reply from s;
    // site will send a list of all remote conflicts
    update SG_T based on reply information;
    if (new dependencies emerged){
        //propagate changes to pre-ordered transactions
        propagate SG_T to pre-ordered transactions;
    }
}
// this step is performed by the parallel thread
// Serialization Graph Update Thread
wait until SG_T does not contain any incoming edges for T;

// 3. Commit Phase:
for(s ∈ S*_T){
    send "Commit" to s;
    update SG_T based on reply information;
}
mark T as "committed" in SG_T;
propagate updated SG_T to post-ordered transactions;
terminate

Serialization Graph Update Thread:
while(true) {
    wait for message with SG_T as an update graph
    or  SG_T locally updated;
    if(message arrived) {
        update  SG_T
    }
    if(SG_T changed){
        send SG_T to pre-ordered transactions
        if(SG_T is cyclic and T is victim){
            abort;
        }
    }
}
```

4.2.2. Site Protocol

```
Direct Operations Invocations Thread:
T_conflicts = ∅;
while(true){
    wait for next message m;
    if(m equals directly execute o_i from transaction T){
        execute o_i;
        schedule indirect operation invocation of o_i;
        for(e ∈ Log) {
            if( e.T' ≠ T and (e.o, o_i) ∈ CON){
                T_conflicts = T_conflicts ∪ T;
            }
        }
        add (o_i, T) to Log;
        return T_conflicts;
    } else if(m equals indirectly execute o_i from transaction T){
        execute o_i;
```

```
17          for (entries e ∈ Log) {
18              if(e.T′ ≠ T and (e.o, oi) ∈ CON){
19                  T_conflicts = T_conflicts ∪ T;
20              }
21          }
22          add (oi, T) to Log;
23          return T_conflicts;
24          // prior to commit, T will request updated
25          // information about remote conflicts
26      } else if(m equals update conflicts for T}
27          // sites which contains replicas of data objects
28          // accessed by T;
29          S* := [s1, ... , sn] ;
30          while(si ∈ S* has not replied with remote conflicts){
31              wait;
32          }
33          for (e ∈ Log and e.T = T_updating){
34              // get remote conflicts
35              for(e′ ∈ Log with e'␣≠ e){
36                  if(e′.T ≠ T_updating and (e.o, e′.o) ∈ CON){
37                      T_conflicts = T_conflicts ∪ e.T;
38                  }
39              }
40          }
41      } else if (m contains message: transaction T will commit){
42          T_post = ∅
43          // get all post−ordered transactions
44          for(e′ ∈ Log with e' > e){
45              if(e′.T ≠ T_committing and (e.o, e′.o) ∈ CON){
46                  T_post = T_post ∪ e′.T;
47              }
48          }
49          // T will commit; remove log info
50          for(e ∈ Log){
51              if(e.T = T_committing){
52                  remove e from Log;
53              }
54          }
55      }
56  }
57
58  Schedule Indirect Operations Invocations Thread:
59  // sites which contains replicas of data objects accessed by T;
60  S* := [s1, ... , sn]
61  while(true){
62      wait for next message m;
63      if(m equals indirect operation invocation of oi ∈ T) {
64          for(si ∈ S*){
65              send message to si: indirectly execute oi;
66          }
67      else if(reply received from si (as a log entry ei)){
68          for(e ∈ Log with e ≠ ei){
69              if(e.T ≠ Ti and (e.o, ei.o) ∈ CON){
70                  T_conflicts = T_conflicts ∪ Ti;
71              }
72          }
```

```
73          return T_conflicts;
74      }
75  }
```

4.3 Re:GRIDiT – 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 [25]. 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 commit projection remains acyclic.

We enforce correct schedules by guaranteeing (i) recoverability and (ii) serializability. The latter means that no cycles in the committed projection [26] 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 1. *Algorithms 4.2.1 and 4.2.2 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 4.2.1, 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 4.2.1, 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 2. *Algorithms 4.2.1 and 4.2.2 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 subcases:

   (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 3. *Algorithms 4.2.1 and 4.2.2 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 4.2.1, 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 4.2.1 line 32). Moreover, it will inform T_p about this order (Algorithm 4.2.1, 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 4. *Algorithms 4.2.1 and 4.2.2 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 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 2 all local and remote serialization orders are the same.

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

Theorems 1 - 4 together prove that the Re:GRIDiT protocol ensures global correctness by guaranteeing both recoverability and serializability in a distributed environment and in the absence of a global coordinator.

5 Conclusions and Outlook

We are faced nowadays with the need for more and more complex computations on huge amounts of data that take place in distributed infrastructures as, for instance, the Grid. There exists naturally the need to ensure a correct execution and guarantee the success of such computations, which can be achieved by enforcing transactional guarantees on their executions. To the best of our knowledge the currently available concurrency protocols do not deal with replicated data in distributed environments without global coordinator. Yet, this is a vital requirement for a large range of applications. In this paper, we propose the first algorithm that solves all these problems are that hides the presence of replicas to the applications and, most importantly, that provides provably correct transactional execution guarantees without any global component.

Nevertheless many open question remain for future research. Although many victim selection strategies exist, several question remain open and certainly require future investigation: i.) how can we optimize the victim selection strategy by exploiting the infrastructure, for example, by taking advantage of the load of the sites, and ii.) how can dynamic placement and deployment of replicas be seamlessly added to the protocol.

References

[1] 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.

[2] 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.

[3] D. Louis Collins, Johan Montagnat, Alex P. Zijdenbos, Alan C. Evans, and Douglas 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.

[4] Web Services Transaction Specifications. http://www.ibm.com/developerworks/library/specification/ws-tx/.

[5] Klaus Haller, Heiko Schuldt, and Can Türker. Decentralized coordination of transactional processes in peer-to-peer environments. In *CIKM*, pages 28–35, 2005.

[6] Can Türker, Klaus Haller, Christoph Schuler, and Hans-Jörg Schek. How can we support Grid Transactions? Towards Peer-to-Peer Transaction Processing. In *CIDR*, pages 174–185, 2005.

[7] Zhengwei Qi, Xiao Xie, Baowen Zhang, and Jinyuan You. Integrating X/Open DTP into Grid Services for Grid Transaction Processing. In *FTDCS '04: Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04)*, pages 128–134, Washington, DC, USA, 2004. IEEE Computer Society.

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

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

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

[11] SRB: The Storage Resource Broker. http://www.sdsc.edu/srb/.

[12] Esther Pacitti, Pascale Minet, and Eric Simon. Replica Consistency in Lazy Master Replicated Databases. *Distrib. Parallel Databases*, 9(3):237–267, 2001.

[13] Todd Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool. Replication, consistency, and practicality: are these mutually exclusive? In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 484–495, New York, NY, USA, 1998. ACM.

[14] Fuat Akal, Can Türker, Hans-Jörg Schek, Yuri Breitbart, Torsten Grabs, and Lourens Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In *VLDB*, pages 565–576, 2005.

[15] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[16] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[17] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive replication in peer-to-peer systems. In *In The 24th International Conference on Distributed Computing Systems*, pages 360–369, 2004.

[18] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Data currency in replicated DHTs. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 211–222, New York, NY, USA, 2007. ACM.

[19] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):177–190, 2002.

[20] Fuat Akal, Heiko Schuldt, and Hans-Jörg Schek. Grid-Enabled Data Replication for Digital Libraries with Freshness and Correctness Guarantees. In *3rd VLDB Workshop on Data Management in Grids (DMG 2007), Vienna, Austria*, 2007.

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

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

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

[24] G. Weikum and G. Vossen. *Transactional Information Systems - Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, San Francisco, California, 2002.

[25] 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.

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