How can we support Grid Transactions? Towards Peer-to-Peer Transaction Processing*

Can Türker Klaus Haller Christoph Schuler Hans-Jörg Schek

ETH Zurich, Institute of Information Systems, CH-8092 Zurich, Switzerland
{tuerker|haller|schuler|schek}@inf.ethz.ch

Abstract

Today, we witness a merger between Web services and grid technology towards an open grid service infrastructure that especially satisfies the demands of complex computations on huge volumes of data. Such applications are specified as combinations of services and are executed as workflow processes. While transactional support was neglected for (business) workflows, in the grid domain we observe not only a more general usage of workflow technology but also a stronger awareness of transactional guarantees. The rigid database notions of atomicity and isolation are however not suited for composite services in grid applications because of their complexity and duration. Beyond, the level of abstraction in the grid is far above database pages such that two-phase commit combined with two-phase locking as the state-of-the-art for distributed transactions is not adequate. Rather, compensation of services, restarting services, and invoking alternative services are needed. In this context many questions are open. How does the infrastructure detect and handle conflicts? What happens if a service is unavailable? Can we locally decide whether a distributed execution of transactions is globally correct? In this paper, we tackle some of these questions and sketch an approach to ensuring globally correct executions of transactional processes without a global coordinator.

*Papers on "Transactions" are no longer desired in our mainstream conferences. Nevertheless, we are convinced that much work is still needed and useful, especially with respect to newer computing paradigms.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 2005 CIDR Conference

1 Introduction

Motivated by many e-science and e-business applications that operate on massive amounts of data and run very complex computations anytime and anywhere, many new distributed computing paradigms have been emerged in the recent years with the goal to provide a common infrastructure with nearly unlimited storage and computational capabilities. These main trends are "Web Services", "Grid Infrastructures", and "Peer-to-Peer Computing".

Web services provide a set of standard technologies, such as WSDL (Web Service Description Language) [35], UDDI (Uniform Description Discovery and Integration) [2], and SOAP (Simple Object Access Protocol) [27], to describe, discover, and invoke *any* kind of services in a networked environment. The platform-independent definitions of these technologies simplify the composition of services to offer new value-added services [24]. The service-oriented architecture allows services to be integrated into an overall architecture as a service with a well-defined interface contract. BPEL4WS (Business Process Execution Language for Web Services) [5] is used to specify such compound services. *Workflow management systems* like IBM's MQSeries [20] support the integration of Web service calls into workflow processes. Since these systems follow a centralized architecture consisting of dedicated workflow engine(s), their scalability is limited. Modern process management systems like IBM's Process Choreographer [30] still rely on a central database for process instances.

Grid systems, such as the Globus Toolkit [14], provide a wide support for effective resource management and load balancing. Essentially, these systems maintain the available resources of a grid and assign tasks to the least loaded peers. In addition, it is even possible to install new services on the grid in case a bottleneck is detected. However, these systems lack a sophisticated support for combining several service calls into processes. This is because they act more like a UDDI repository focusing on optimal routing of service requests.

Peer-to-peer systems like Gnutella [15] support a higher degree of autonomy and control over the services the peers utilize. Each participating peer acts both as a client and as a server depending on the semantics of the application. Each peer can initiate requests and can respond to requests from other peers in the network. The ability to directly communicate with other peers avoids *central* servers, and thus provides the basis for optimal scalability.

Today, we witness the merger of these technologies towards an open grid services infrastructure/-architecture (OGSI/OGSA) [11], which allows for dynamically sharing and amplifying any kind of computing resources. Many efforts are underway in the Global Grid Forum (GGF) [13] to document "best practices", implementation guidelines and standards for these technologies, which are subsumed under label "The Grid" [10]. Moreover with the merger, we observe a shift towards a more general and especially commercial usage of the grid to build highly-scalable virtual organizations [12].[1] A number of international projects have been initiated in this direction. EGEE (Enabling Grids for E-science in Europe) [8], for example, is a large ongoing European project with the goal to develop a new grid infrastructure for "all" applications, from digital libraries to e-health and e-science. Other examples are US GRIDS Center [29], UK Grid Support Centre [28], D-Grid (German Initiative) [7], NAREGI (Japanese Initiative) [21], to name just a few.

With the more general and particularly commercial usage of the grid, a lot of questions arise. In this paper, we focus on the aspect of correct concurrent execution of grid applications:

- How do transactions fit into grid environments?
- How does the grid infrastructure decide what to do if dependencies to other concurrent transactions exist?
- What information will lead to a decision for a compensation of a previously executed service and what is the right compensation?
- Does partial rollback make sense?
- Can we locally decide whether a distributed execution of transactional workflows is globally correct?
- What happens if a service or more generally a peer that provides services is unavailable?

In this paper, we provide answers to some of these questions. Specifically, we present protocols for concurrent transactional processes that ensure globally correct executions without involving a global coordinator. A key assumption in these protocols is that they sit on top of database transactions. We do not touch database transactions but rather use them as basic service. We try to exploit the service semantics and in this aspect we keep our tradition and follow the directions of nested and composite transactions [33, 1]. We also adapt old ideas from serialization graph testing and distributed deadlock detection [22] to the new environment.

The considerations presented in this paper belong to the *hyperdatabase*[2] project and concentrate on decentralized concurrency control. The main idea of our approach is that dependencies between transactions are managed by the transactions themselves. A core aspect is that globally correct executions can be achieved even in case of incomplete knowledge by communication among dependent transactions and the peers they have accessed. The proposed protocol relies on a decentralized serialization graph, where each peer and each transaction maintain a local serialization graph. While the serialization graph of a peer reflects the dependencies among the transactions that invoked services on that peer, the serialization graph of a transaction includes the dependencies in which the transaction is involved.

The rest of the paper is organized as follows: Section 2 discusses the notion of a transaction within the grid context. Section 3 sketches our decentralized approach to concurrency control and recovery in grid systems and discusses its basic assumptions and limitations. Section 4 concludes the paper with an outlook on open research problems.

2 Towards Grid Transactions

Database systems ensure correct executions of applications under concurrency and failures situations using the concept of a transaction. A transaction consists of a sequence of operations working on database objects like tables. Traditionally, a database transaction is associated with the ACID properties [16] which provide a set of well-understood and established execution guarantees. The acronym ACID stands for atomicity, consistency, isolation, and durability. Atomicity requires that all or none of operations of a transaction are executed. Consistency states that the execution of a transaction leads from consistent to another consistent database state. Isolation demands that concurrent transactions execute as if they were executed

[1] "Grids Deployed in the Enterprise" is the title of the twelfth GGF, which was held in Brussels, Belgium during 20-23 September 2004. This title emphasizes the mentioned shift towards commercializing the grid.

[2] The concept of a hyperdatabase combines the best concepts of the three distributing computing directions, i.e., Web services, grid computing and peer-to-peer networks, to provide a highly-scalable and efficient infrastructure that especially supports transactional guarantees at the level of processes. For more details about the vision of hyperdatabases and some special issues like peer-to-peer execution of processes, we refer to [23, 26].

in isolation. Durability requests that the effect of the execution of a transaction persists.

These ACID requirements however seem to be unnecessarily rigid for grid environments. Grid transactions differ from traditional database transactions in the following ways:

- Grid transactions are composed of service calls. As depicted in Figure 1, these service calls can be executed by different peers of the grid. In contrast to the operations of database transactions, services provide a much higher-level of abstraction such that their semantics can be exploited, for instance, to increase the degree of parallelism by performing semantic concurrency control [31].

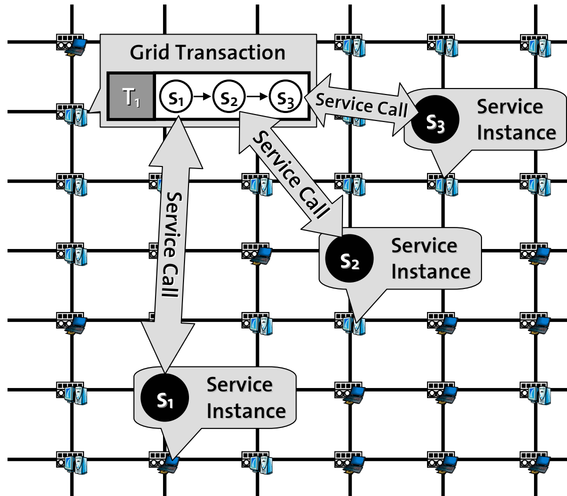

Figure 1: Transaction over the Grid

- A large number of grid transactions consist of operations that represent long-running complex computations on huge data volumes. Hence, according to the atomicity criterion, a failure in the computation would lead to the rollback and restart of many computation steps. Here, a more flexible way of recovery is needed which avoids such complete rollbacks.

The health sector in all western societies faces the problem of exploding costs. One approach is to reduce costs by automatic business processes. Since there is no single, uniform information system in the health sector, process technology only becomes beneficial if processes are supported which span over activities provided by different peers. In reality, processes consist of computationally expensive activities like computer-aided diagnosis, large-scale knowledge discovery, and biological simulation. Besides, processes might include long-running activities like patient-customized analysis or accurate therapy planning. All these activities are often combined with rather short activities like payments of doctor or health insurance bills. Using process technology in the health sector requires that the underlying data is neither corrupted nor inconsistent. Otherwise, life and health of patients is in danger. Thus, transactional execution guarantees are required.

Another example for the application of transactions in the grid environment is the parallelization of numerical algorithms under multi-level transaction control. As shown in [9], some algorithms allow a higher degree of parallelism.

State-of-the-art process management systems like *IBM Websphere Process Choreographer* [30] provide transactional guarantees with an important restriction based on the type of the processes:

- Short-running processes are realized by distributed database transactions.

- Long-running processes run as a set of decoupled local transactions.

While the atomicity of short-running processes is enforced by blocking protocols like the two-phase-commit (2PC) protocol, the atomicity of long-running processes is guaranteed by executing compensating transactions. In case a step of a short-running process fails, the WebSphere Application Server rolls back the entire distributed transaction. In case of long-running processes, a failure of a step is handled by semantically undoing the previously committed steps of that process. In this case, the WebSphere Process Choreographer runs compensation transactions on the corresponding databases.

In case of short-running processes, isolation is implicitly handled by mapping the processes onto distributed database transactions. These distributed transactions are processed by the underlying J2EE infrastructure of the WebSphere Application Server. The isolation of concurrent long-running processes, however, is not supported by the WebSphere Process Choreographer since the distributed locking protocol of the underlying infrastructure would block the system for a significantly long period. Therefore, the task of ensuring isolation is left to the application level, and thus it is not supported.

The durability of a process execution is achieved by using persistent queues and storage. Consistency relies on the correct specification of the processes. This can be supported by verification of the correctness of the process at deployment time.

During transaction processing every *pessimistic* protocol intercepts the execution of a transaction in order to perform a global checking. In case of a locking protocol, such as the two-phase locking (2PL) protocol, the underlying infrastructure requests a lock for a transaction and waits until it has received the lock. In a loosely-coupled environment like in the grid domain, such a global checking will force a *synchronous* communication between the corresponding peer and the

other peers or a central component, respectively. The application of a pessimistic locking protocol is therefore much more expensive than using an *optimistic* variant of the serialization graph testing protocol. The latter performs the global (cycle) checking at the end of each transaction. Clearly, this checking procedure has a linear run-time complexity. Albeit this is too expensive in traditional database settings, in the context of long-running processes in grid networks, the cycle checking is no longer *expensive compared to the operation execution cost.* Interestingly, previous research has not seen this fact. Serialization graph testing has been used only in theory to demonstrate serializability theory. Moreover, note that a synchronous checking has to be performed only once per transaction. Using incremental conflict replication, necessary graph data will be available at commit time. In this way, typically no further synchronization will be needed at commit time.

3 Decentralized Concurrency Control and Recovery for Grid Transactions

Before we present our proposal for a completely distributed, peer-to-peer style processing of transactions over the grid, we first summarize the basic assumptions of our simple grid system model and briefly introduce our transaction model.

3.1 The Grid System Model

We start our considerations with the following basic grid system model, which provides a general distributed computing environment:

- The grid consists of peers that are able to directly communicate with (all) other peers of the grid.
- As depicted in Figure 2, which zooms into a single peer, each peer provides a set of services. These services can be invoked using the service interface of the corresponding peer. These services are executed as local database transactions.
- To ease the discussion, we assume that the peers are independent in the sense that conflicts can only appear among service invocations on the same peer. In particular, services are not replicated on different peers. We later discuss in Subsection 3.6 how to overcome these restrictions.
- When a new service is registered at a peer, conflicts to other services at this peer must be stated in this registration step. As in semantic concurrency control [31], conflicts are defined based on the semantics of the services. Usually, these conflicts are stored in a service-level conflict matrix. As a refinement, as done in the notions of backward and forward commutativity [31], the definition of a conflict may also take the arguments and return values of service invocations as well as the current state of the underlying data(base) values into account.
- A grid transaction is a multi-level transaction, as already sketched in Figure 1. The leaves of a grid transaction correspond to invocations of basic services. Thus, a grid transaction can be seen as a compound service with transactional guarantees.

To perform semantic concurrency control and recovery for grid transactions, compensation of services must be defined beside the definition of conflicts between services. In addition, conflicts must be detected and solved to ensure serializable executions. And finally, in case of necessary rollbacks compensation must be initiated to ensure semantic atomicity.

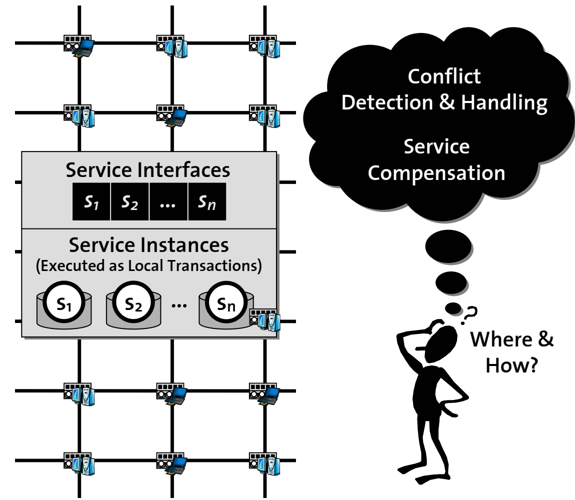

Figure 2: Basic Grid System Model

The question is how to include these extensions into the grid system model sketched above. A straightforward — but not convincing — approach would be to introduce dedicated peers with special grid services. One dedicated peer could act as a central coordinator which performs concurrency control and recovery based on complete global knowledge, for instance, in form of a global lock table or a global serialization graph. Each peer would contact this coordinator for each service invocation. Using its complete knowledge about global conflicts, the coordinator would determine whether the execution of this service invocation would cause a non-serializable schedule.

In large-scale grids with thousands of peers, the central coordinator will become very soon the bottleneck of the system. Moreover, it will be a single point of failure. Therefore, this approach is not suited for grid systems. In Subsection 3.3, we sketch a more promising and innovative approach which completely distributes the task of global coordination in a peer-to-peer style over the grid.

3.2 Grid Transactions as Transactional Processes

We use transactional processes [25] to execute transactions over the grid. The model of transactional processes generalizes traditional database transactions especially with respect to the demand for more flexibility and a higher level of abstraction. A *transactional process* comprises a set of service invocations which are executed in a specified order. Every transactional process can be used in a more complex transactional process as a single service invocation. In this sense, every transactional process represents a compound service. Besides sequential execution of service invocation, transactional processes allow parallel execution and alternative execution paths. Based on the structural constraints of the model of transactional processes, every transactional process is guaranteed to terminate in the well-defined final state. In other words, there is always an execution path that eventually terminates. In the remainder of this paper, we will use the term "*transaction*" as short-hand for "*transactional processes*".

The notion of a schedule is fundamental for defining a correctness criterion for concurrent executions of transactions. A schedule reflects the temporal order in which the services of the transactions were executed. It also specifies the order of all *conflicting* services that appear in it.

Usually, correctness is defined based on the notion of *serializability* [3]. A schedule is *serializable* (correct) if and only if there is a serial execution of same transactions in which the same conflicts occur. Since serial executions of transactions are correct per definition, serializable schedules are correct, too. In theory, serializability of a schedule is checked using a serialization graph. The nodes of the graph correspond to transactions of the schedule, while the edges correspond to conflicts between these transactions. A schedule is serializable if and only if its serialization graph is acyclic [3]. Multi-level serializability [32] provides us a correctness criterion for semantic concurrent control over composite transactions.

3.3 Ensuring Global Serializability without a Global Coordinator

In the following, we present an approach that 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 peers of the grid in a uniform way, i.e., it does not assume any dedicated peers.

The first question we have to answer is how to produce globally serializable schedules without complete global knowledge. For this, consider the following fundamental *commit rule*: A transaction is not allowed to commit if it is dependent on another active transaction. A global coordinator who knows about all transactions in the system would be able to delay or reject the commit request of a transaction if this transaction depends on another active one.

However, we can enforce the commit rule in a completely decentralized way if all the transactions in the system can decide on their own whether or not they are allowed to commit [18]. For this decision, the transactions do not require full global knowledge about all conflicts in the system. Rather, it is sufficient that at commit time the corresponding transaction knows about all conflicts it is involved in. If the transaction depends on any other transaction, it has to delay its commit until all these transactions have committed.

The next question is how can a transaction get all the necessary information. For this, a transaction must collaborate with peers and other transactions. At service invocation time, a peer determines the local conflicts using its local log and returns the information about conflicts to the transaction together with the result of the service invocation. In this way, each transaction knows exactly about the transactions it depends on.

Before we present the protocol, we summarize the necessary extensions of the basic grid system model we have introduced before:

- Since we assume that conflicts cannot occur between services of different peers, the conflict information can be *partitioned* over the peers. Each peer maintains a conflict matrix indicating which service invocations cause conflicts. Conflicts are defined by the peer (service provider) based on the semantics and invocation parameters of the services. Using its conflict matrix, each peer is able to autonomously detect conflicts between service invocations of different transactions.

- Each peer provides a compensation service for each of its services to perform compensation of local service invocations. The operations of the compensation services strongly depend on the semantics of the original service. The compensation might also be a "do nothing" service.

- Each peer contains a local log where it stores information about the invocation of local services. Using this information, a peer can derive conflicts between transactions that have invoked services on this peer.

- Each transaction manages its own serialization graph which comprises the conflicts in which the transaction is involved in. Essentially, the graph contains at least all conflicts that cause the transaction to be dependent on other transactions. This partial knowledge is sufficient for a transaction to be able to decide whether it is allowed to commit.

- Each peer provides a transaction execution environment which allows for invoking services within grid transactions on any peer of the grid.

Figure 3 illustrates the constitutes of a grid peer, which we have explained above.

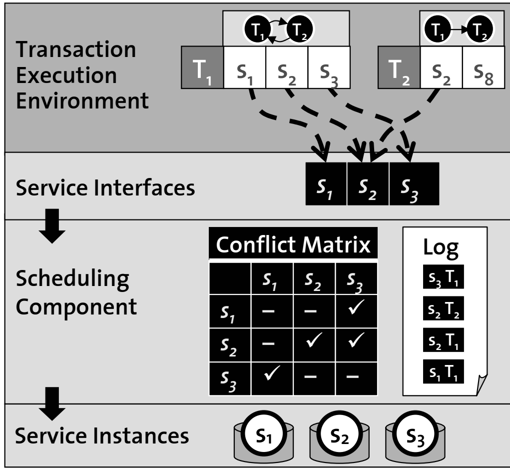

Figure 3: Constitutes of a Grid Peer

Since the envisioned protocol requires collaboration among transactions and peers, it consists of two parts: One part is running on each peer, the other part for each transaction. The peer part of the protocol performs the following:

1. In case of a service invocation, the peer logs the service invocation, executes the service, and determines all conflicts (if any) using the local log and the local conflict matrix. Finally, it sends back the result of the service invocation to the invoking transaction together with a complete list of conflicts that have occurred. This list of conflicts contains all service invocations of other transactions at this peer that are in conflict with the current invocation.

2. In case of a commit message from a transaction, the peer provides a list with all transactions that are dependent on this committing transaction. This information is needed to inform the dependent transactions about the commit of this transaction. The dependent transactions might already wait for the commit of this committing transaction in order to commit themselves.

The part of the protocol that runs for each transaction consists of three phases:

1. *Service execution phase*: Following its specification, the transaction invokes services on certain peers in an optimistic manner without requesting any locks. As described in the peer protocol above, the corresponding peers execute the requested service invocations, detect conflicts, and return them back to the transaction with the results of the service invocations.

2. *Validation phase*: As soon as the transaction has executed all of its specified services, it validates whether or not it is allowed to commit by checking its local serialization graph. If there is no incoming edge to the corresponding node, i.e., if the transaction does not depend on any other active transaction, then it enters the commit phase. Otherwise, it waits until the corresponding active transaction have committed, i.e., the corresponding edges have disappeared from the local serialization graph.

3. *Commit phase*: The transaction commits and informs the peers on which it has invoked services about its commit. According to the peer protocol, the peers determine the conflicts with all transactions that depend on this committing transaction and send the information back to the committing transaction. The latter inserts these conflicts into its local serialization graph and informs the dependent transactions about its commit. This is necessary because these transactions might wait for this commit in order to safely commit as well subsequently. The deletion of nodes and edges from the local serialization graphs in case of a commit of another transaction is handled by an independent thread of the protocol which is triggered by an incoming commit message from a committing transaction.

To sum up, transactions invoke services without determining on the spot the corresponding effects on the serialization graph. Nevertheless, at least prior to the commit, a validation is performed that checks whether the transaction has been executed correctly and is therefore allowed to commit. This is closely related to well-established optimistic concurrency control protocols like backward-oriented concurrency control [19] as well as to serialization graph testing protocols as proposed in [6].

The protocol incorporates some nice characteristics that boost its performance: It allows a decoupled propagation of conflicts such that the transaction knows about the dependencies as soon as possible. At commit time, the transaction has the necessary information to perform or delay its commit. In contrast to a two-phase commit protocol, the transaction need not to initiate a communication with all other transactions. It simply checks its local serialization graph and performs a commit if there is no active transaction on which it depends on. Otherwise, if there is such a transaction, the transaction will wait until it receives a commit message from the corresponding transaction. In this way, the communication overhead needed at commit time is reduced dramatically.

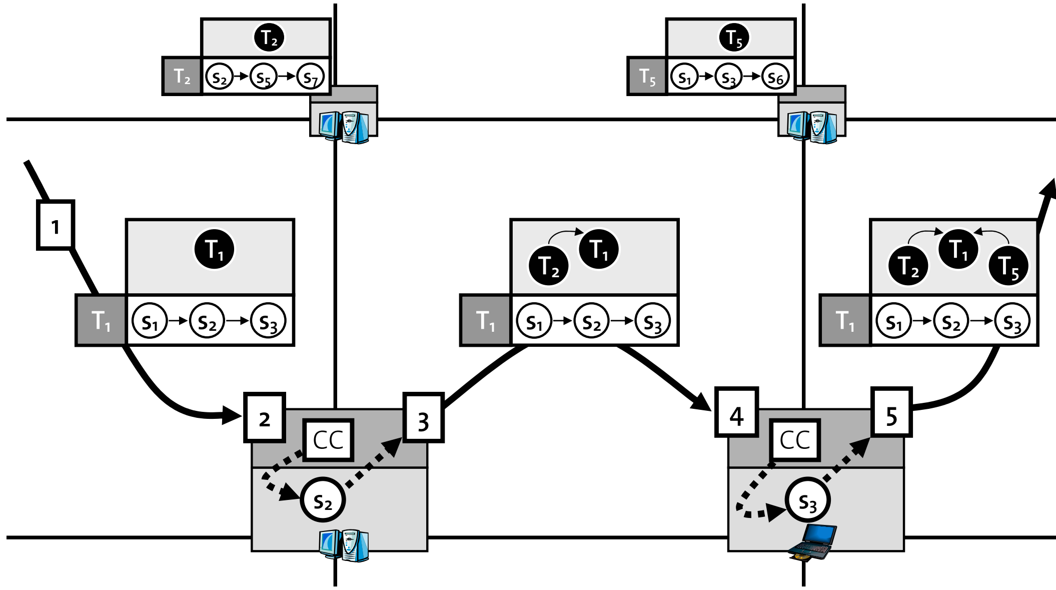

Figure 4: Peer-to-Peer Processing of Transactions over the Grid

Furthermore, the protocol allows a peer-to-peer processing of grid transactions. As sketched in Figure 4, a transaction moves with its context from peer to peer until it has executed all its specified services. In the depicted example, the transaction T_1 has already executed a service s_1 on any peer of the grid. The execution of this service did not cause a conflict. Therefore, the local serialization graph of T_1 still has no edges. In the second step, T_1 arrives at a peer where it invokes service s_2. Let this invocation lead to a conflict with another transaction T_2. The peer detects this conflict based on its local log and informs T_1 about it. T_1 updates its local serialization graph and goes over to execute the next service according to the process definition.

Of course, the protocol could also be implemented in traditional way such that the transactions do not move from peer to peer. Rather, they remotely invoke services at the corresponding peers.

3.4 Distributed Cycle Detection

Besides guaranteeing that a transaction only commits if it is not dependent on any other active transaction, the protocol has to detect and solve cyclic dependency situations: In case of cyclic dependencies transactions hinder each other to commit. This will not change without intervention. Since cyclic dependencies might be caused by conflicts among two or more transactions that are executed on different peers, neither a single peer nor a single transaction can detect cycles by only relying on their incomplete local knowledge.

Two main directions of approaches known from the area of distributed deadlock detection can adapted to solve the isolation problem mentioned above:

1. *Timeout approaches*: A transaction, which wants to commit but is hindered by one or more transactions on which it depends, sets a timeout interval. Within this time period, the transaction must be able to commit. Otherwise, the transaction assumes to be involved in a cyclic dependency and hence aborts. In this way, cycles are eventually broken. However, the timeout approach comes to the price of unnecessary delays and many unnecessary aborts.

2. *Graph exchange approaches*: Both problems of the timeout approach can be avoided by exchanging the serialization graph information among the transactions:

   (a) *Full replication*: Every change in a local serialization graph is propagated to all transactions. This implies that all transactions have full knowledge, and thus will detect only existing cycles. Clearly, replicating the whole serialization graph results in a huge number of messages, and moreover distributes a lot of information that is not needed to fulfill the given task.

   (b) *Partial replication*: A more sophisticated approach is based on the idea to propagate only the part of the local serialization graphs that is actually needed for a certain transaction to detect existing cycles. Again, two different approaches can be used to distribute this information:

      i. A transaction sends its local serialization graph only to those transactions that are included in its graph. From a global point of view, the virtual global serialization graph is partitioned

i. into disconnected subgraphs (partitions) such that each subgraph contains all the necessary information to detect existing cycles.

ii. Following the path-pushing approach [22], the communication overhead can be reduced further if the transactions only propagate changes in the local serialization graph to transactions on which they depend. These changes will then be transitively distributed following the dependency paths.

Clearly, *synchronous* updates of the local serialization graphs are not appropriate for any kind of distributed environment due to performance reasons, as shown in [17]. Therefore, the update propagation has to be performed in a decoupled manner in either approach.

We run first experiments as a proof of concept on six computers of our computer cluster. All cluster nodes were IBM Blade Center HS20 equipped with Dual Intel Xeon 3.2GHz, 2GB Ram, 1 Gigabit Fiber network and Microsoft Windows 2003 Server. Five clients together run 100 transactions in parallel. The clients were using J2SE 1.3.1, IBM Classic VM with JITC. The clients communicated with each other via Java-RMI. On the sixth computer, we run IBM WebSphere Application Server 5.1.1 as a service provider. This equipment allowed to measure without considering load or bandwidth problems.

We varied the conflict probability by changing the number of services offered by the WebSphere host — traditionally, this is done by varying the number of data objects. The clients invoked the services via Web services that were mapped to EJB session beans reading and writing back a 16 MB data object. To simulate long-running transactions, each of the transactions not only consisted of 8 to 12 Web service invocations, but additionally got a penalty of two additional seconds waiting time on the server as well as on the client side.

Figure 5 shows the results for the following three approaches:

1. 2PL/1PC with centralized deadlock detection as an implementation of a traditional system infrastructure,
2. our path-pushing approach with partial rollback, and finally
3. a conflict-free environment.

At a first glance, it is surprising that the throughput decreases by increasing the number of services implying a decrease of the conflict probability. However, this is a consequence of the fact that the application server consumes more time to manage the increasing number of entity beans. Thus, it is more interesting what happens with the curves of the S2PL and the path-pushing approach. In case of many services (7000-10000), i.e., low conflict probability, the throughput all three approaches is practically identical. For higher conflict probabilities, the path-pushing approach outperforms the locking-based approach by far, for instance, by a factor of 3.9 for 3000 Services and by a factor of 2.8 for 4000. More details and further experiments are documented in [4].

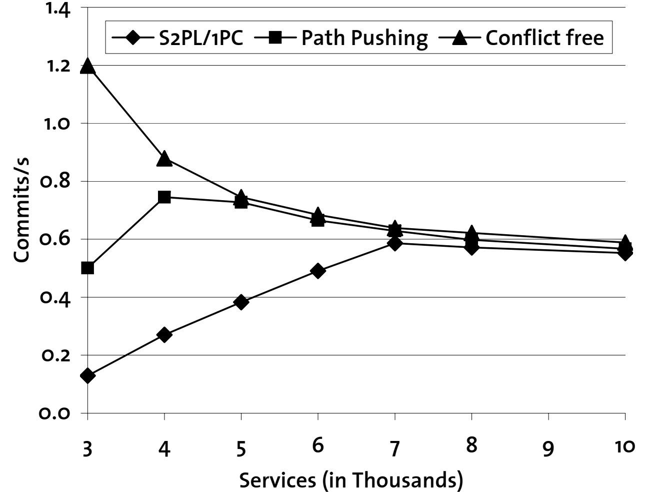

Figure 5: Throughput

3.5 Recovery based on Partial Rollback

Cascading aborts appear when a transaction has to rollback due to the rollback of a transaction it depends on. Not surprisingly, commercial database systems use locking-based protocols, which avoid cascading aborts. In case of a failure, a locking-based protocol rolls back only one transaction completely. Other active transactions are not affected by this rollback. Such approaches are suitable for short-living transactions, as they usually appear in traditional databases applications. Grid transactions, on the other hand, usually are long-running. Undoing the entire transaction would mean to loose a lot of work, especially due to the effect of cascading aborts, because at least *all* transactions involved in the cycle have to be rolled back. Therefore, workflow management systems and process engines do not implement recovery strategies for isolation failures.

However, we address this problem in this paper. We combine our non-blocking serialization graph testing approach with *partial rollback*. Partial rollback means to rollback a transaction in case of isolation failures by invoking compensation services *only* to a step at which the cycle in the serialization graph disappears. Then, the transaction resumes its execution at this step. Hence, only one victim has to rollback until the

incoming and outgoing edges contributing to the cycle cease to exist.

We illustrate this using the following sample schedule (same services are assumed to be in conflict):

$$s_A^{T_1}, s_B^{T_1}, s_C^{T_1}, \underbrace{s_D^{T_1}, s_D^{T_2}}_{T_1 \rightarrow T_2}, \underbrace{s_E^{T_2}, s_E^{T_1}}_{T_2 \rightarrow T_1}$$

In this schedule, the transactions T_1 and T_2 cause a cycle. With complete rollback, all service invocations of both transactions would be compensated because of cascading aborts. In case of partial rollback, T_2 could be the victim. Besides T_2's own service invocations, only $s_E^{T_1}$ would have to be compensated.

Obviously, partial rollback is beneficial because it minimizes the effects of cascading aborts and thus allows for a higher throughput. Choosing the "proper" victim by using graph information now becomes a parameter for further tuning.

Using partial rollback for grid transactions requires orchestrating the recovery on the different peers of the grid, because we do not want to rely on a centralized component due to the disadvantages mentioned before. Basically, a grid transaction involved in the cycle is selected as victim. This transaction then invokes compensation services in the opposite ordering of the invocation of the originally services. For invoking compensation services, the transaction sends *Compensate-Service* messages to the corresponding peers. The following rules sketch the interaction between transactions and peers:

Peer behavior. A peer receiving a *Compensate-Service(s_u)* message executes the compensation service s_u^{-1} only if there are no obstacles. An *obstacle* s_o is a service executed after s_u which is in conflict with s_u^{-1}. The peer collects all obstacles that have to be compensated before the service s_u can be compensated; let S be this collection. Finally, the peer sends an *Obstacle(S)* message back to the transaction. However, to prevent that in the following additional obstacles appear, the peer will not execute services raising additional obstacles with respect to s_u^{-1}.

Transaction behavior/Obstacle message. When a transaction asks a peer to compensate a service invocation, the peer might reject this and return a set of obstacles to the grid transaction. The transaction then sends a *Compensate-Obstacle* message to the grid transactions which executed the obstacles and waits until all have done so. Then, the grid transaction sends again a *Compensate-Service* message. This time, the peer can compensate s_u since all previously existing obstacles are removed and new ones have been prevented to appear.

Transaction behavior/Undo-Transaction message. When a transaction receives such a message and is not already in the recovery mode, it stops the forward execution and compensates at least until the step specified in the message. As soon as the transaction compensated a service which is an obstacle for another transaction, it informs the latter.

To perform these tasks, the transactions and peer require certain information. The peers have to log all service invocations and require a conflict matrix — both are also needed for the concurrency control task. In addition, a set of variables is needed for logging all service invocations that the transactions intend to compensate but cannot do up to now because of obstacles. On the other side, transactions must know on which peers they have invoked services, which steps have to be compensated in case of a rollback, which obstacles prevent to immediately compensate a service, and, finally, which of its service invocations are obstacles for other transactions.

Though first experiments show a high increase in the transaction throughput based on partial rollback, several questions are still open:

1. Compensating the victim grid transaction completely is a straight-forward approach, but how can the "border" be determined as accurate as possible to further reduce the number of compensated services?

2. Under which circumstances emerges the starvation problem [34] and how can we deal with it?

3. How can we optimize the victim selection by exploiting the various parameters of the grid, as the load of the peers for example?

3.6 Transaction Processing in the Presence of Peer-Spanning Conflicts

The approach presented up to now assumes that conflicts only occur between local services. Grid systems, however, must be able to deal with replicated services, as it is often the case for computational services. Therefore, we generalize our approach in the following such that it also supports concurrency control and recovery in case of services that are replicated over a set of peers. We even go a step further in allowing conflicts among services hosted on different peers.

Providing a correct execution in this generalized setting does not require that all peers are aware of all possible conflicts. In addition to all local conflicts, the conflict matrix of each peer only has to be extended by the conflicts involving a service provided locally. Peer-spanning conflicts are determined and specified by the administrators of the corresponding peers. To detect a peer-spanning conflict $s_x \rightarrow s_y$, the following extensions are needed:

1. The conflict matrix of the peer that provides s_y must contain a corresponding entry that reflects the conflict $s_x \rightarrow s_y$.

2. Since conflicts are detected based on local log entries, the local log of the peer that provides s_y must contain all occurrences of s_x. That is, this peer must be informed about such occurrences by the peer that executes s_x.

Consider Figure 6 which shows an example for a peer-spanning conflict. Peer P1 provides the services s_1, s_2, and s_3, while peer P2 provides s_4 and s_5. The matrices show the conflicts of the peer's services. The semantics of the matrices is as follows: Executing, for example, s_4 conflicts to a prior execution of s_5. The opposite execution order, in contrast, is allowed. In this example, it is assumed that s_4 conflicts to any prior execution of s_2.

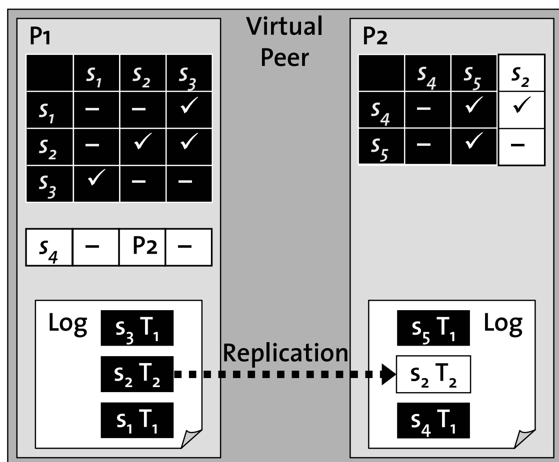

Figure 6: Peer-spanning Conflict in a Virtual Peer

The white color for the column of the peer-spanning conflict in P2's conflict matrix is used only for illustration purposes. In fact, the underlying conflict detection routine does not distinguish between local and peer-spanning conflicts. For each new local log entry, it simply checks whether or not the corresponding service invocation causes a conflict.

In contrast, the white row associated with P1's conflict matrix has a different semantics. It belongs to an additional data structure which stores the information about the peers that must be informed when the corresponding service is invoked.

Both extensions depend on the current system configuration. They have to be consistent even in case of that the set of available services and peers changes dynamically. This consistency can be ensured by the meta-data replication functionality of the underlying grid infrastructure, for example, by using a publish-subscribe-based replication as presented in [26]. Suppose the service s_2 is no longer provided by peer P1, then the corresponding column is eliminated from P2's conflict matrix (when there is no other replica of s_2 in the grid. Any update on the global configuration will cause an update of such replicated information. In this way, both peers P1 and P2 become aware of all "peer-relevant" (global) conflicts.

In addition to the conflict matrix, which depends on the relatively static system configuration, a complete service invocation log has to be provided at each peer. Beside all information about local service invocations, this log now has to include also the invocations of conflicting services on other peers. In the example above, the log of peer P2 must replicate all invocations of s_2 on peer P1, since a local execution of s_4 conflicts with a prior execution of s_2. The update of the two invocation logs must be synchronized. This can be realized by either a blocking synchronous communication, or again by an optimistic protocol.

An alternative approach would be to define a *virtual peer*. A virtual peer subsumes a set of peers that share at least one common conflict. No service within a virtual peer conflicts to any other service outside the virtual peer. In this way, a virtual peer can be seen as a single peer with respect to concurrency control and recovery. Beside the synchronization of the invocation logs, a virtual peer can also be realized by defining a single peer. In this way, the coordination for the virtual peer is centralized, and the virtual peer conceptually degrades to a usual peer.

3.7 Transaction Processing in Disconnected Networks

The complete distribution of the transaction execution moreover allows to run transactions on disconnected partitions of the grid network (see Figure 7). In contrast to a centralized coordination, this is possible since the protocol for distributed concurrency control primarily relies on local peer information. However, the propagation of serialization graphs requires that transactions are connected to each other if they are involved in a conflict. Thus, if no transaction migrates from one partition to another, concurrency control works in spite of disconnection.

If a transaction migrates from one partition to the next, the partitions have to be connected during the migration of the transaction. Furthermore, a connection must be established at least at commit time while resolving dependencies if there are any conflicts with services from peers of a different partition. For instance if in the example in Figure 7 T_2 would depend on T_1, the commit of T_2 would be deferred until the corresponding partitions are connected again.
This blocking situation can only be omitted by relaxing the isolation level: The transaction can commit with an option to later compensate or reconcile the complete transaction. However, correctness can no longer be guaranteed in case of such cascading dependencies.

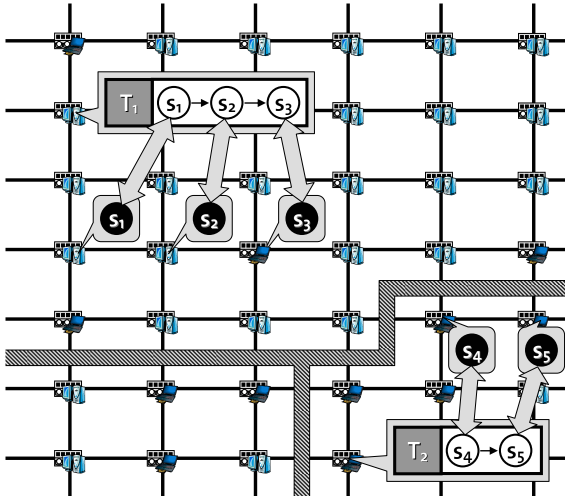

Figure 7: Disconnected Transaction Processing on a Partitioned Grid

While transactions can run on disconnected partitions, a virtual peer should reside completely within one partition, because the local invocation logs have to be updated synchronously. Therefore no services are allowed to be executed while the members of a virtual peer are disconnected.

4 Conclusions and Outlook

The merger of service-oriented and grid technologies opens the possibility to develop infrastructures that provide nearly unlimited computation and storage resources. In this context, grid transactions are a key concept for correct concurrent executions of grid applications that especially invoke data services. However, traditional database transaction processing techniques cannot be applied directly to grid transactions, because the latter usually are long-running and the conflict probability is low in the grid. In this paper, we presented a new approach to processing transactions over the grid. Our approach is a unique combination of known techniques used for a new purpose:

1. The recoverability criterion, which demands that a transaction must not commit if it depends on an active transaction, is used to allow transactions to check with their local knowledge alone whether or not they are allowed to commit.

2. Serialization graph testing as a concurrency control protocol was condemned for years, because it imposes a high overhead, and thus it was seen as useless in practice. However, for long-running grid transactions, the overhead induced by a graph cycle checking is no longer a hurdle because this overhead is not significant anymore with respect to the long running time of grid transactions.

3. Path-pushing approaches are known for years for deadlock detection, but we now use it for serialization graph testing purposes. In this way, correctness of global executions can be ensured without maintaining a global serialization graph which reflects all conflicts that occur in the grid.

4. Partial rollback and repeatable activities are also well-known concepts from the area of workflows. Our innovation is to use it for isolation failures to address the problem of cascading aborts.

Additionally, we sketched two important conceptual extensions to this core model for grid transaction processing. Firstly, replication of services and the underlying data is important for the grid context, but needs the concept of virtual peers or corresponding replication of the conflict and log information. Secondly, large networks are never 100% stable and also grid networks contain more and more mobile devices. Thus, the aspect of disconnection was included into our work.

Nevertheless many open questions are left for further research: For example, the aspect of costs has to be taken into consideration when compensating service invocations. Also mixing different isolation levels is a hot topic, especially since commercial systems support this for database transactions (which are used in the grid context to execute basic services). The best victim selection strategy is still an open problem, which gets more exciting since the grid provides additional tuning parameters. Another interesting topic is the support of ad-hoc grid transactions, which represent individualized processes that only run once or a few times. For such kinds of transactions it seems to be not reasonable to be pre-installed on the corresponding peers of the grid. In general, a deeper investigation of the effect of grid dynamics, such as disconnected peers/services and dynamically changing grid partitions, on concurrency control and recovery is required.

References

[1] G. Alonso, S. Blott, A. Feßler, and H.-J. Schek. Correctness and Parallelism in Composite Systems. In *Proc. of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'97, May 12–14, 1997, Tuscon, Arizona, USA*, pages 197–208. ACM Press, New York, 1997.

[2] Ariba, IBM, and Microsoft. UDDI Technical White Paper. `http://www.uddi.org`.

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

[4] O. Biderbost. Peer-to-Peer Transaction Processing in a WebSphere Application Server Environment. Master's thesis, ETH Zurich, Switzerland, 2004. (In German).

[5] BPEL – Business Process Execution Language for Web Services, Version 1.0. `http://www.ibm.com/developerworks/webservices/library/ws-bpel/`.

[6] M. Casanova. *The Concurrency Control Problem for Database Systems*. Springer-Verlag, 1981.

[7] D-GRID (Initiative for Grid-based E-Science Framework in Germany). http://d-grid.de/.

[8] EGEE: Enabling Grids for E-science in Europe. http://eu-egee.org/.

[9] A. Feßler and H.-J. Schek. A Generalized Transaction Theory for Database and Non-Database Tasks. In P. Amestoy, P. Berger, M. J. Daydé, I. S. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *Euro-Par'99, Proc. of the 5th European Conf. on Parallel Computing, Toulouse, France, August 31–September 3, 1999*, Lecture Notes in Computer Science, Vol. 1685, pages 459–468. Springer-Verlag, Berlin, 1999.

[10] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, 2 edition, 2004.

[11] I. Foster, C. Kesselmann, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Global Grid Forum 2002*, 2002. http://www.gridforum.org/ogsi-wg.

[12] I. Foster, C. Kesselmann, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3):123–130, 2001.

[13] The Global Grid Forum (GGF). http://www.ggf.org/.

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

[15] Gnutella RFC. http://rfc-gnutella.sourceforge.net.

[16] J. Gray. The Transaction Concept: Virtues and Limitations. In C. Zaniolo and C. Delobel, editors, *Proc. of the 7th Int. Conf. on Very Large Data Bases, VLDB'81, Cannes, France, September 9–11, 1981*, pages 144–154. IEEE Computer Society Press, Los Altos, CA, 1981.

[17] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In H. V. Jagadish and I. S. Mumick, editors, *Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Quebec, Canada*, ACM SIGMOD Record, Vol. 25, No. 2, pages 173–182, ACM Press, June 1996.

[18] K. Haller, H. Schuldt, and H.-J. Schek. Transactional Peer-to-Peer Information Processing: The AMOR Approach. In M.-S. Chen, P. K. Chrysanthis, M. Sloman, and A. B. Zaslavsky, editors, *Proc. of the 4th Int. Conf. on Mobile Data Management, Melbourne, Australia, January 21–24, 2003*, Lecture Notes in Computer Science, Vol. 2574, pages 356–362. Springer-Verlag, Berlin, 2003.

[19] H. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. In A. L. Furtado and H. L. Morgan, editors, *Proc. of the 5th Int. Conf. on Very Large Data Bases, VLDB'79, October 3–5, 1979, Rio de Janeiro, Brazil*, pages 350–350, IEEE Computer Society Press, 1979.

[20] IBM MQSeries Workflow. http://www.ibm.com/software/ts/mqseries/workflow/.

[21] NAREGI (National Research GRID Initiative of Japan). http://www.naregi.org/index_e.html/.

[22] R. Obermarck. Distributed Deadlock Detection Algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, 1982.

[23] H.-J. Schek, H. Schuldt, C. Schuler, and R. Weber. Infrastructure for Information Spaces. In *Advances in Databases and Information Systems, Proc. of the 6th East-European Symposium, ADBIS'2002, Bratislava, Slovakia, September 2002*, Lecture Notes in Computer Science, Vol. 2435, pages 23–36. Springer-Verlag, Berlin, 2002.

[24] M. Schmid, F. Leymann, and D. Roller. Web Services and Business Process Management. *IBM Systems Journal*, 41(2):198–211, 2002.

[25] H. Schuldt, G. Alonso, and H.-J. Schek. Concurrency Control and Recovery in Transactional Process Management. In *Proc. of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'99, May 31–June 2, 1999, Philadelphia, Pennsylvania*, pages 316–326. ACM Press, New York, 1999.

[26] C. Schuler, R. Weber, H. Schuldt, and H.-J. Schek. Peer-to-Peer Process Execution with OSIRIS. In *Service-Oriented Computing — ICSOC 2003, First International Conference, Trento, Italy, December 15–18, 2003, Proceedings*, Lecture Notes in Computer Science, Vol. 2910, pages 483–498. Springer-Verlag, Berlin, 2003.

[27] SOAP – Simple Object Access Protocol. http://www.w3.org/TR/SOAP/.

[28] UK Grid Support Centre. http://www.grid-support.ac.uk/.

[29] US GRIDS Center. http://grids-center.org/.

[30] IBM WebSphere Application Server Enterprise Process Choreographer. http://www7b.software.ibm.com/wsdd/zones/was/wpc.html.

[31] W. E. Weihl. Commutativity-based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computer*, 37(12):1488–1505, 1988.

[32] G. Weikum. Principles and Realization Strategies of Multilevel Transactions Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[33] G. Weikum and H.-J. Schek. Concepts and Applications of Multi-level Transactions and Open Nested Transactions. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–553, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[34] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2001.

[35] WSDL – Web Service Description Language. http://www.w3.org/TR/wsdl/.