In: Proceedings of the 14^{th} ACM International Conference on Information and Knowledge Management (CIKM 2005), pages 36-43, Bremen, Germany, November 2005. ACM Press.

Decentralized Coordination of Transactional Processes in Peer-to-Peer Environments *

Klaus Haller[1] Heiko Schuldt[2] Can Türker[3]

[1] AWK Group Zurich, post@klaus-haller.net

[2] UMIT, Information & Software Engineering Group, Hall in Tyrol, Austria, heiko.schuldt@umit.at

[3] Functional Genomics Center Zurich, Uni / ETH Zurich, tuerker@fgcz.ethz.ch

ABSTRACT

Business processes executing in peer-to-peer environments usually invoke Web services on different, independent peers. Although peer-to-peer environments inherently lack global control, some business processes nevertheless require global transactional guarantees, i.e., atomicity and isolation applied at the level of processes. This paper introduces a new decentralized serialization graph testing protocol to ensure 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 transactional process is equipped with partial knowledge that allows the transactional processes to coordinate. Globally correct execution is achieved by communication among dependent transactional processes and the peers they have accessed. In case of failures, a combination of partial backward and forward recovery is applied. Experimental results exhibit a significant performance gain over traditional distributed locking-based protocols with respect to the execution of transactions encompassing Web service requests.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Concurrency, Transaction processing

General Terms

Measurement, Performance, Reliability

Keywords

DSGT, Decentralized Coordination, Global Correctness, Peer-to-Peer Communication, Transactional Processes, Partial Rollback.

1. INTRODUCTION

With the proliferation of e-business technologies, service-oriented computing is becoming increasingly popular. Access to data and documents is provided by *services* which can range from simple read/ write operations on data items to complex business functions like booking a trip. An important challenge is to combine service invocations into a coherent whole by means of *processes* [13, 3]. Workflow and process technologies support this kind of service composition, usually provided by sophisticated system infrastructures like IBM's WebSphere [6]. However, such systems require a global coordinator (or a set of such coordinators replicated for performance reasons). While this can be easily enforced for well-established business interactions, it is no longer true when interactions rather follow an ad-hoc style.

In peer-to-peer (P2P) environments, each peer provides a set of services that can be composed to processes. These processes might run over several peers. An important task in such environments is to ensure a globally correct execution of these processes, i.e., to provide atomicity and isolation applied at the level of processes. This demands for a concurrency control and recovery technique that respects the P2P style of communication between the system components and that is able to scale to large networks of peers.

Conventionally, isolation and atomicity are enforced using a locking protocol like the strict two-phase locking (2PL) in combination with a global commit protocol like the two-phase commit (2PC) [1, 11]. Such protocols are usually applied to short living transactions and are state-of-the art in application domains which allow centralized approaches. However, for P2P environments concepts from distributed transaction processing are required, i.e., S2PL is combined with a distributed deadlock detection protocol like [10, 12, 7]. Other options are optimistic protocols or timestamp ordering protocols, respectively. Optimistic protocols, such as proposed in [8], can be applied in P2P environments without modifications. They execute transactions completely and check directly before the commit whether the transaction is allowed to commit. Thus, optimistic approaches come along with a high number of rollbacks when the duration of transaction execution is high. Timestamp ordering protocols are often used in distributed environments since they do not require coordination of different resources [1]. Instead, each transaction is associated with a timestamp reflecting its entrance into the system. The ordering of executed service invocations of different transactions on each peer must reflect this order. Thus, a high number of transactions are unnecessarily aborted. In distributed environments, the additional problem of global time and clock synchronization arises.

In this paper, we present a new protocol for ensuring concurrency control and recovery especially in P2P environments. Essentially, the protocol ensures globally correct executions without involving a global coordinator. The main idea of the protocol is that dependencies between transactions are managed by the transactions themselves. A core aspect is that globally correct executions can be

*This work was funded in part by the Swiss National Science Foundation within the project MAGIC.

CIKM'05, October 31–November 5, 2005, Bremen, Germany.

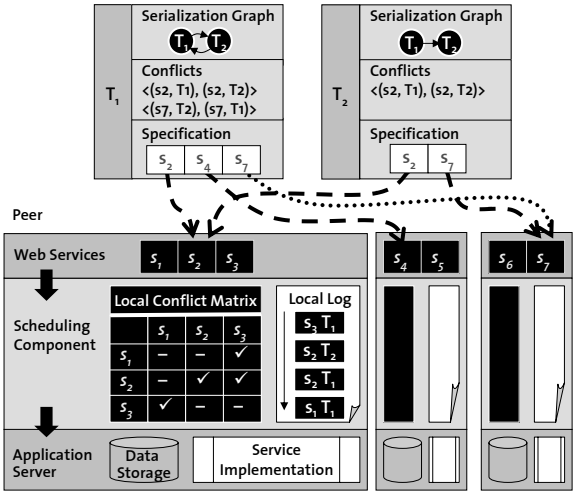

Figure 1: System Model

achieved even in case of incomplete knowledge by communication among dependent transactions and the peers they have accessed. The protocol relies on a decentralized serialization graph, where each peer logs local conflicts and each transaction maintains a local serialization graph. While the local conflict information 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.

Since synchronous updates are not appropriate for any kind of distributed environment due to performance reasons [4], the update of the local serialization graphs is performed in a lazy manner. In consequence, the serialization graphs will not necessarily be up-to-date. If at commit time a transaction is able to deduce out of its local serialization graph that does not depend on another active transaction, it is allowed to commit. Conversely, if a transaction detects a cycle in the serialization graph, the cycle has to be resolved by rolling back one or more transactions involved in this cycle. Here, combining partial backward and forward recovery allows to significantly reduce the amount of work needed to recover from such a failure.

The paper is organized as follows: Section 2 introduces our decentralized serialization graph (DSGT) protocol for ensuring concurrency control and recovery in P2P environments. Section 3 presents results we achieved from experiments with our protocol. Section 4 reviews related work and Section 5 concludes.

2. THE DSGT PROTOCOL

2.1 System Model

As illustrated in Figure 1, we assume a P2P network where each *peer* P_i offers a set of services $O^{P_i} = \{s_1^{P_i}, s_2^{P_i}, \ldots s_{n_i}^{P_i}\}$. The services of a peer can be invoked within transactions $T_k = (O_{T_k}, <_{T_k})$ using the service interface of that peer. In the following, O_{T_k} denotes the set of services to be invoked by transaction T_k and $<_{T_k}$ the partial order defined over O_{T_k}.

A transaction may fail due to several reasons. To ensure atomic executions, the effects of the transaction's service invocations must be compensated. This compensation is done by invoking compensation services in reverse order (cf. [13]). Following usual practice in semantic concurrency control, we also assume that the peers provide for each service s_j they offer an inverse service s_j^{-1} that

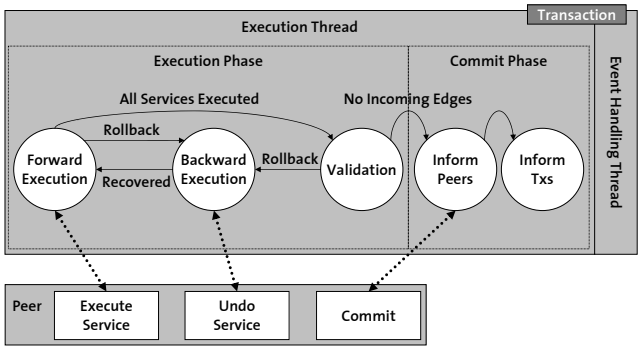

Figure 2: States and Transitions in the DSGT Protocol

semantically undoes the effect of the invocation of s_j. Note that the effects of an inverse service strongly depend on the semantics of the original service. It might also be an "empty service".

For correctness, we rely on the criterion *conflict preserving serializability* [1]. Following [15], a schedule (the services invoked by the transactions of the system and the order between them) is correct, if and only if the serialization graph of the schedule is acyclic. In this case, the schedule is called *serializable*. A serialization graph contains a dependency between two transactions, if there is at least a pair of service invocations of both transactions that is in conflict, i.e., changing the order of these service invocations results in a different final system state or different system outputs (a formal definition for conflicts between semantically rich operations can be found in [15]).

Enabling the peers to detect conflicts between service invocations performed by them requires to equip each peer with its local conflict matrix. This matrix contains information which services of that peer pairwise conflict (and under which conditions). To concentrate on the main aspects of DSGT, we assume here that service invocations on different peers are not in conflict. However, results presented in [14] are applicable to remove this restriction. In addition, each peer has to store in its local log which service invocations of which transactions it has executed. Using this information together with the local conflict matrix, a peer detects conflicts between local service invocations of different transactions. Note that the peers do not maintain serialization graphs but only local logs.

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

But reasoning whether a transaction is allowed to commit is not sufficient. Additionally, the system must be able to recover from failures. In what follows, we use the notion of *recoverability* [1] as criterion for correct failure handling. It is important to note that our system model does not require a component that maintains a global serialization graph.

2.2 Idea and Overview of DSGT

The decentralized serialization graph test (DSGT) protocol relies on the following observations:

- Dependencies between transactions can be managed by the transactions themselves.

Algorithm 1: Peer Protocol

```
while true do
    wait for next message m;
    switch message type of m do
        case T_invoking invokes service s_i
            execute service s_i;
            set T*_conflicting := ∅;
            foreach e ∈ Log do
                if e.T ≠ T_invoking ∧ (e.service, s_i) ∈ CON then
                    T*_conflicting := T*_conflicting ∪ T_invoking;
                end
            end
            add (s_i, T_invoking) to Log;
            return T*_conflicting;
        case T_c commits
            // collect all dependent transactions
            T*_post := ∅;
            foreach e ∈ Log with e.T = T_c do
                foreach e' ∈ Log with e' > e do
                    if e'.T ≠ T_c ∧ (e.service, e'.service) ∈ CON then
                        T*_post := T*_post ∪ {e'.T}
                    end
                end
            end
            // remove all log information of committing transaction T_c
            foreach e ∈ Log do
                if e.T = T_c then
                    remove e from Log;
                end
            end
    end
end
```

Algorithm 2: Transaction Protocol

```
S_T := [s_1, ..., s_n]; // services to be invoked by T
P_T := {}; // peers on which T invoked services
SG_T := {}; // local serialization graph of T

Main Execution Thread:
// 1. invoke services and update serialization graph
foreach s_i ∈ S_T do
    invoke s_i at an appropriate peer p and add p to P_T;
    wait for reply from p;
    update SG_T based on reply information;
    propagate changes to pre-ordered transactions T's (if SG_T is
    changed);
end
// 2. wait until all pre-ordered transactions have committed
wait until there is no incoming edge to node T in SG_T;
// 3. inform all peers such that they can clean up their logs
foreach p ∈ P_T do
    send 'commit' to peer p;
    update SG_T based on reply information;
end
mark T as 'committed' in SG_T;
propagate updated SG_T to post-ordered transactions;
terminate;

Event Handling Thread:
while true do
    if new graph message SG_new arrived then
        let SG_T^red the reduced version of SG_T ∪ SG_new ;
        propagate SG_T^red to pre-ordered transactions (if SG_T^red
        changed);
    end
    if SG_T changed ∧ SG_T cyclic ∧ T is victim then
        abort;
    end
end
```

- Globally correct execution can be achieved even in case of incomplete knowledge and in the absence of a global coordinator by communication among dependent transactions and the peers on which these transactions have performed service invocations.

Following these observations, DSGT guarantees that a transaction only commits after all its pre-ordered transactions have committed. A transaction receives the information about its pre-ordered transactions from the peers where it has invoked services. Essentially, each peer attaches a list of conflicting services and their associated transactions to the results of each service invocation. The transaction maintains these dependencies in its local serialization graph and is then able to decide autonomously whether or not it is allowed to commit. If at commit time a transaction detects that there is an active transaction on which it depends, it waits until it receives information about the commit of the other transaction. Thus, it is part of the protocol that each transaction informs all its post-ordered transactions about its commit. The transaction receives the information about its post-ordered transactions at commit time from the peers on which it has invoked services.

It is important to stress that DSGT is an optimistic variant of a distributed serialization graph testing protocol. Services are executed without checking for conflicts, i.e., conflicts are detected afterwards and – in contrast to the pessimistic centralized serialization graph testing – (cascading) rollbacks may be needed to resolve cyclic dependencies. To reduce the recovery costs, DSGT applies *partial* rollback by executing compensation services until the point where the cyclic dependencies disappear. Then, DSGT continues the forward execution of the services from that point on.

Figure 2 illustrates the states of the DSGT protocol with a special focus on the part running on the transaction side.

The part of the protocol that runs on each peer reacts on requests from transactions. Each peer awaits requests from transactions:

- 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 file and the local conflict matrix. Finally, it sends back to the invoking transaction the result of the service invocation 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.
- If compensation of a service invocation is requested, the peer executes the corresponding compensation service or informs the transaction that other transactions have to compensate service invocations cascadingly. Note that a peer does not distinguish between regular services and compensation services. From the point of view of a peer, both are just services.
- If a transaction T_m wants to commit, it informs all peers on which it has invoked services. These return the list of post-ordered transactions on this peer. This information is needed to inform the dependent transactions about the commit (these dependent transactions might already wait for the commit of T_m in order to commit themselves).

Algorithm 2.2 defines the part of the protocol that runs on each peer.

The part of the protocol that runs on each transaction consists of two threads. The proactive execution thread is always in one of the following states:

Forward Execution: The transaction invokes services according to its specification.

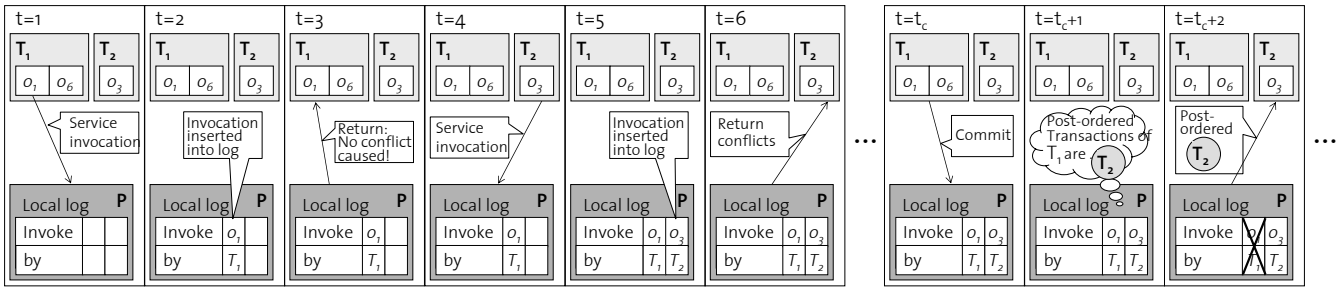

Figure 3: Service Invocation Procedure

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

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

Inform Peers: The transaction informs all peers on which it has invoked services about its commit. Thereby, the transaction not only gets the information from these peers about its post-ordered transactions, it additionally prevents that in future any peers will return a conflict where this transaction is involved in.

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

The following state transitions can take place:

Forward Execution → Validation: If the transaction has executed successfully all specified services, it changes to the validation state.

Validation → Backward Execution: If the transaction detects that it is involved in a cycle and that it is the victim for rollback, it changes to the backward execution state.

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

Backward Execution → Forward Execution: As soon as all required service invocations are compensated, the transaction changes to forward execution again.

Validation → Inform Peers: This transition occurs when the validation yields that the local serialization graph contains an incoming edge from an active transaction.

Inform Peers → Inform Txs: This transition happens as soon as all corresponding peers have responded to the transaction.

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

In the following subsection, we show how DSGT ensures global correctness.

2.3 Ensuring Global Correctness

The main execution thread of the transaction protocol consists of three phases. Phase 1 is the execution phase. In this phase, a transaction invokes services in an optimistic manner without requesting any locks (cf. $t = 1$ and $t = 4$ in Figure 3). Then, the peers execute the services according to Algorithm 2.2. The peers determine the emerging conflicts ($t = 2$ resp. $t = 5$) and return them to the transaction ($t = 3$ resp. $t = 6$). As soon as the transaction has executed all services, the main execution thread enters the validation phase. The transaction now has to wait until it does not (or no longer) depend on any other active transaction. As soon as this condition is fulfilled, the main execution thread enters the commit phase, in which the peers are informed about its commit. Additionally, the transaction informs its post-ordered transactions about its commit. This is necessary since these transactions might wait for this commit in order to commit as well. The serialization graph update thread of the other transactions receives this information and changes the corresponding local serialization graph accordingly.

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 whether it is therefore allowed to commit. This is closely related to well-established optimistic concurrency control protocols like backward-oriented concurrency control [8] and to serialization graph testing protocols [16]. As in all other cases, transactions are allowed to commit in DSGT only if they do not depend on an active transaction. Transactions get this information from the peers they invoke services on. We thus can state, that *no transaction commits before all its pre-ordered transactions have committed*, which guarantees a serializable schedule (formal proofs are presented in [5]).

Secondly, an important aspect of the commit phase is that *transactions willing to commit eventually succeed when all pre-ordered transactions have committed* (the proof can also be found in [5]). Consider again Figure 3. T_1 does not know about the conflict with T_2. However, in order to be able to commit, T_2 must be informed when T_1 commits. Therefore, since the dependency with T_1 is not known to T_2, we cannot require the latter to query T_1 for its state. Rather, T_1 has to actively notify all peers it has accessed during its execution about its commit ($t = t_c$). Each peer checks for relevant conflicts ($t = t_c + 1$) and returns this information to the issuing transaction (T_1) before removing the entries of the committing transaction from the log (at time $t_c = t + 2$).

2.4 Cycle Detection

A transaction involved in a cycle must detect this. To detect a cycle, a transaction must have the relevant conflict information. Therefore, transactions have to exchange their local knowledge on

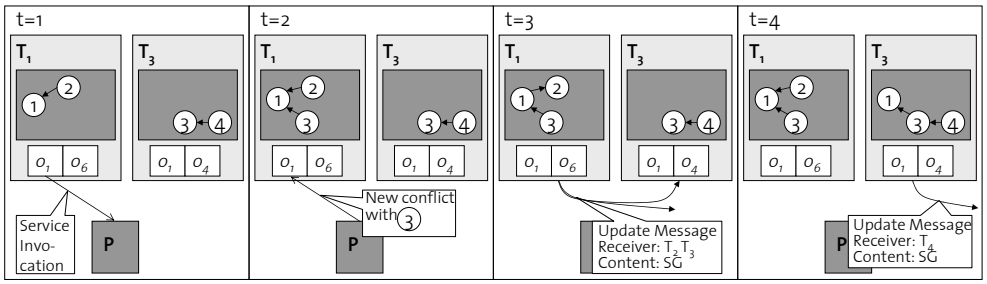

Figure 4: Cycle Detection

conflicts. Since synchronous updates of local serialization graphs are not appropriate for any kind of distributed environment due to performance reasons [4], the update has to be performed in a lazy manner. In consequence, the serialization graphs will not necessarily be always up-to-date.

A cycle in the serialization graph implies the following:

1. None of the involved transactions can commit due to cyclic dependencies.
2. The cycle will not disappear without any intervention.
3. Cycles might be caused by conflicts of more than two transactions executed on different peers. So neither a peer nor a transaction is able to detect cycles based on their local knowledge alone.

The implementation of the information exchange for detecting cycles covers three aspects: i.) If a transaction causes a new conflict, it updates its local graph and propagates the graph to its pre-ordered transactions. ii.) A transaction uses a graph received from another transaction to update its local serialization graph. If this leads to changes, it propagates its updated graph to its pre-ordered transactions (Certainly, other approaches work theoretically as well, e.g., distributing conflict information to all other transactions in the system or to all other ones of the same partition of the serialization graph. The other extreme – no communication at all between transactions – would correspond to a timeout heuristic: A transaction not being able to commit assumes after some time to be involved in a cycle and rolls back. However, due to lack of space, we concentrate in this paper on the path-pushing approach which turned out to be the most efficient one in our experiments). iii.) If the transaction detects a cycle and the victim selection strategy selects itself as the victim, it aborts.

This approach is heavily inspired by distributed deadlock detection algorithms, especially by path pushing approaches such as presented in [10]. Of course, the semantics of serialization graphs and wait-for-graphs are different. Figure 4 illustrates the graph propagation mechanism of DSGT. The figure shows the local graphs of the transactions T_1 and T_3. At time $t = 1$, T_1 invokes a service on the peer, which causes a conflict with T_3. The peer returns this information at $t = 2$. T_1 updates its local serialization graph with this information ($t = 3$). Then, T_1 propagates its graph to the pre-ordered transactions T_2 and T_3. At $t = 4$, T_2 receives this message. It updates its local graph. After updating its graph, T_3 propagates the changes to T_4.

2.5 Partial Rollback

DSGT uses partial rollback to reduce the costs of rollbacks. Basically, a transaction does not roll back completely, but only to the point at which the (isolation) failure is resolved, i.e., where the cyclic dependencies have disappeared. This concept is applicable for all kinds of protocols, but it is especially useful for reducing the effect of cascading aborts which may appear in DSGT. Figure 5 illustrates this. There are five transactions whose service invocations lead to a cycle in the serialization graph. Assume that T_1 is chosen as victim. Using complete rollback implies that undoing $s_a^{T_1}$ requires to undo also transaction T_2 because of the conflict $(s_a^{T_1}, s_a^{T_2})$. But then, not only $s_d^{T_2}$, also $s_b^{T_2}$ and $s_e^{T_2}$ have to be compensated requiring to undo also T_5 and T_3. Choosing T_2 or T_3 instead of T_1 as victim would lead to the same result. So, obviously, a cycle in case of serialization graph testing implies to rollback all transactions forming the cycle plus all post-ordered transactions.

Partial rollback may reduce this drawback: Choosing T_1 as the victim to be rolled back completely implies that T_2 compensates $s_a^{T_2}$ because of the conflict. This requires T_2 to compensate also $s_d^{T_2}$ because it has been executed after $s_a^{T_2}$. However, additional compensations are not necessary and especially $s_b^{T_2}$ remains untouched, so that T_3 and T_5 do not have to rollback. Thus, this example shows how the avalanche of cascading aborts can be stopped by using partial rollback.

To express in a schedule how far a transaction has to be compensated, we introduce the partial rollback operator r_{T_v,s_b}. It denotes that service invocations should be rolled back until (and including) service s_b of T_v.

Following the ideas of the unified theory of concurrency control and recovery [15], an abort is replaced by a sequence of compensation service invocations of the associated forward execution in reverse order. This is called *expansion*. In here, we assume perfect commutativity behavior [15]. The expanded schedule S' comprises (1) the "old" service invocations, (2) the service invocations of the victim transactions which have to be undone, and (3) all cascading compensation service invocations.

Partial rollback can be used for recovering a schedule from an isolation failure. The following *rule* states how and where to insert the partial abort operator in the schedule[1] (in what follows, we denote the service invocations of T_1 as $s_{1_1} \dots s_{n_1}$.): Let $S = (O, <)$ be the schedule, in which the transactions $T_1 \dots T_n$ form a cycle. Let T_v be the victim transaction selected out of $T_1 \dots T_n$. Then the resulting schedule $S' = (O', <')$ is constructed as following:

$$O' = O \cup \{r_{T_v,s_{1_1}}\} \quad \text{and} \quad <' = < \cup \{s_{n_v} < r_{T_v,s_{1_1}}\}$$

A schedule containing a partial rollback operator is correct, if the expansion of this schedule can be *reduced* to a serial one using an arbitrary sequence of the following transformation steps:

1. Commutativity Transformation: Two service invocations of

[1]This includes the orthogonal problem of victim selection. However, our experiments show that choosing the youngest transaction is usually the best approach.

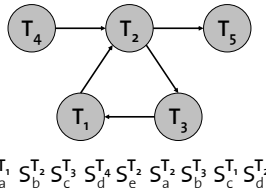

Figure 5: Cyclic Serialization Graph

different transactions might be exchanged, if and only if there is no other service invocation between them establishing a transitive order between the two.

2. Undo Reduction Transformation: Two service invocations can be removed together, if and only if (i) the second one is the inverse of the first and (ii) there is no transitive ordering established between them by a third service invocation.

We conclude this section with highlighting an important property of our operator placement strategy. If a set of transactions forms a cycle in the serialization graph, the rollback operation placement leads to an acyclic serialization graph after the expansion and reduction of the schedule (proof see [5]).

3. EXPERIMENTAL EVALUATION

We have evaluated DSGT in an application server environment. The experimental setup consists of client hosts, on which transactions run, and one application server representing a peer.[2] The application server follows a three-tier architecture. The upper layer is the Web Container which constitutes a full-fledged HTTP server. The Web Container hosts Web service servlets handling SOAP service calls on the client hosts. Every time a SOAP request arrives, the Web service servlet in the Web container invokes a stateless session bean in the EJB container. The EJB container forms the middle layer of the application server. Besides the session beans, this layer also manages persistently stored entity beans. The entity beans are mapped onto a relational database (we have used IBM Cloudscape), which forms the lowest level of the application server. The EJB-based three tier architecture is the most common approach to support a service-oriented environment. Therefore, we have chosen this architecture for the evaluation of DSGT.

On the client side, the transactions run in Java 2 (Standard Edition) Virtual Machines following DSGT and S2PL, respectively (S2PL has been chosen since it is the most commonly used protocol). To have a fair comparison, we implemented S2PL as good as possible (note that the common transaction processing standards like JTS include two-phase-commit but do not address locking of resources at service level). We even realized the deadlock detection in an optimal way by implementing a centralized deadlock detection component. This component checks immediately for cycles as soon as new dependencies emerge in the system. The communication between transactions for graph exchange (serialization graph testing protocol) as well as between transactions and the centralized deadlock detection component (S2PL) is based on Java RMI.

In the experimental evaluation, we varied the *number of services* between 2000 and 10000 to parameterize the conflict probability. The higher the number of services, the less is the conflict probability. On each of the five client hosts we used, there are always 20

[2]Note that our protocol relies on communication between transactions but *not* on communication between peers. Thus, we can evaluate our P2P-approach with only one application server, but with many client servers.

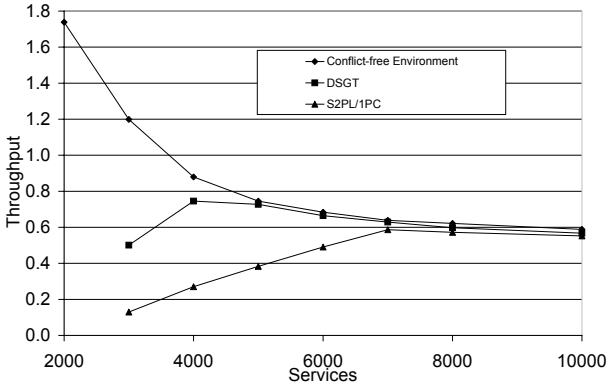

Figure 6: Absolute Throughput

active transactions making a total of 100 *active transactions*. The *length of the transactions* is 8 to 12 service invocations. This value is uniformly distributed.

To simulate complex services, we introduce a *delay on the application server side*. The application server defers the return message sent to the transactions after the service has been executed for two seconds. The *delay on the client side* when receiving a return message from the application server is also set to two seconds to simulate user interaction. The *object size* of the entity beans is 16 MB. In other words: Invoking a service implies that the session bean will call one entity bean such that 16 MB data is read and written back. In case of a deadlock or of a cycle in the serialization graph, the youngest involved transaction is chosen as *victim* and has to compensate completely. Afterwards, to prevent running into the same failure situation repeatedly, a transaction defers the first service invocation by 0 to 20s (uniformly distributed) after changing from backward execution again to forward execution.

The configuration of the client hosts and the application server host has been chosen as follows:

Processor: Dual Intel Xeon 3.2GHz with Hyperthreading
RAM: 2GB
Network: 1 Gigabit Fiber
Operating System: Microsoft Windows 2003 Server
Client JVM: J2SE 1.3.1, IBM Classic VM with JITC
Application Server: IBM WebSphere Application Server 5.1.1

Our first experiment investigates the impact of the conflict probability on the throughput of DSGT and S2PL, respectively. To vary the conflict probability, we have modified the number of services in the system. Figure 6 shows the results of the experiment including measurements for a conflict-free environment. Measurements for S2PL were impossible for 2000 services (high conflict probability) due to many messages which the Java RMI-based Infrastructure was not able to handle.

Since more services imply a lower conflict probability, one would expect that the throughput increases with the number of services. However, the throughput falls in the conflict-free environment dramatically. The explanation for this is that the increasing number of services leads to an increasing number of entity beans. The management of the entity beans bounds the resources of the WebSphere server and thus lowers the response times. To eliminate this effect from the results, Figure 7 shows the throughput of DSGT and S2PL relative to the throughput of the conflict-free environment.

Here, we see that the throughput of S2PL decreases dramatically

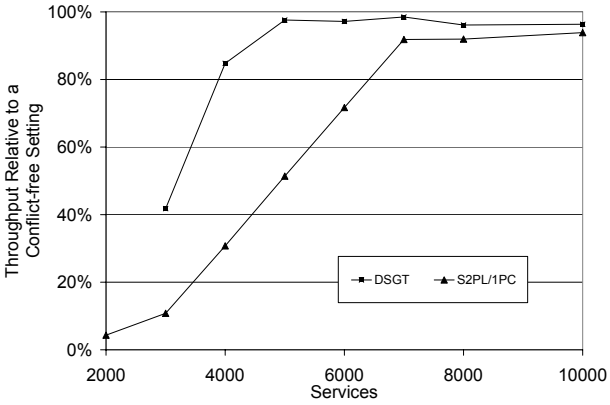

Figure 7: Throughput Relative to a Conflict-Free Environment

for going down to less than 7000 services, whereas DSGT with partial rollback remains on a high level also down to 5000 or 4000 services, although the decrease then is much higher compared to that of S2PL and will drop down to 0 for a high conflict probability. Thus, the experiment shows the superiority of DSGT for *medium* conflict probabilities.

To understand these results better, we have also examined the execution times for transactions in both cases, S2PL and DSGT, for 4.000 services as well as for 10.000 services. The results in Figure 8 show two aspects:

1. The peak of the experiments with 4.000 services is achieved much earlier than in the experiments with 10.000. This proves again the negative impact of a high number of beans on the application server throughput.

2. There is one peak for 4.000 services and S2PL for more than 420s execution time. This states that more than 30% of the transactions need more than 420s to execute (though there is also one peak at 40-60s). This implies that many transactions are blocked by others in case of S2PL. This explains the bad performance of S2PL compared to DSGT.

One might assume that the results might be improved – especially for S2PL – by rolling back a victim transaction not completely but only as far as required, for instance to resolve the cycle. However, we know how much work a transaction has to redo in average (summarized in Table 1).

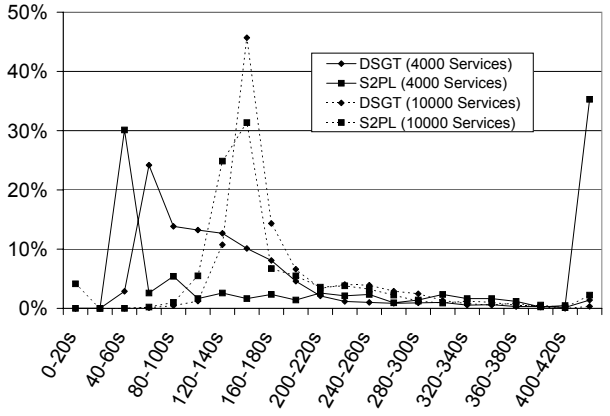

Figure 8: Transaction Execution Time

Services	3000	5000	10000
DSGT	11.18%	0.62%	0.14%
S2PL	6.40%	1.65%	0.09%

Table 1: Percentage of Redo Operations in DSGT and S2PL

Certainly, this value must be doubled to achieve the influence on the overall throughput, because the services have to be compensated before they are invoked again. Thus, for medium conflict probability, this will not change much. It is only relevant for high conflict probabilities. But in this case S2PL is preferred anyway.

Moreover, we investigated the impact of the transaction length on the throughput. We have run experiments where we have chosen the transaction length out of the intervals [4;8], [6;10], and [8;12] (equally distributed). Figure 9 shows the results normalized by the average transaction length for the measurement point. Thus, we have considered that the absolute throughput values are not meaningful, because the longer the transactions are, the more time a single transaction needs even without conflicts. The normalized results show that the throughput of both protocols is more or less equal for short transactions consisting of 4 to 8 service invocations. However, increasing the average transaction length by choosing transaction lengths out of the interval of [6;10] or even [8;12] leads to a tremendous decrease in the throughput of both DSGT and S2PL. Nevertheless, DSGT performs much better than S2PL. For instance, in case of the interval [8;12] the throughput of DSGT is more than 100% higher than of S2PL.

The explanation is simple: Transactions being blocked because they cannot get a lock might itself block subsequent service invocations of other transactions. The DSGT protocol, in contrast, allows the transactions in the same situation to optimistically continue the execution of subsequent service invocations.

Finally, we have compared the partial and the complete rollback case for both DSGT and S2PL. The results in Figure 10 show the impact on partial rollback for cascading aborts.

In case of complete rollbacks, DSGT is only working for very low conflict probabilities: For 6000 services, the throughput is quite low, but it is important to understand that the result marked with (2) appeared. After some time, the throughput falls down to zero making further experiments impossible. In case of 4000 services, the Java RMI problem appeared again (marked with (1)). Thus, this experiment proves that partial rollback is helpful when serialization graph testing is not only used for low but also for medium conflict probabilities.

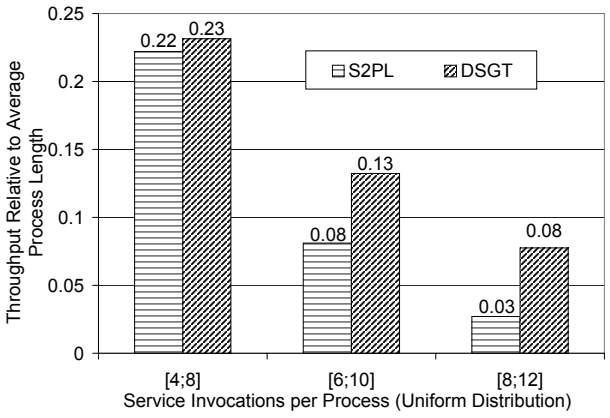

Figure 9: Influence of Transaction Length on Throughput

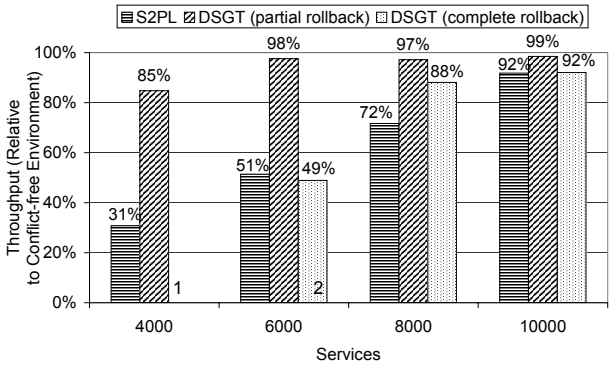

Figure 10: S2PL vs. DSGT (Complete vs. Partial Rollback)

4. RELATED WORK

In principle, protocols developed for distributed and federated database systems are applicable to P2P systems, but all have (some even severe) drawbacks. Optimistic protocols execute transactions without any validation [8]. Therefore, they potentially come along with a large number of rollbacks when the duration of transactions and thus the number of conflicts increases. Distributed variants of optimistic protocols (e.g., [9]) however, stick to a global coordinator, which makes them of limited use for P2P environments.

Our approach fundamentally differs from known distributed serialization graph approaches as presented in [2, 16]. In the latter, the behaviour is distributed, but nevertheless a global graph is maintained. In contrast, in DSGT transactions only maintain parts of the graph knowledge and nevertheless ensure isolation. Due to cycle checking, the complexity of DSGT is linear in the number of transactions in this graph. Compared to a locking protocol like S2PL, this is too expensive for traditional application scenarios. Therefore, serialization graph testing was used in the past only as a formal method to explain serializability theory. Interestingly, our experiments have shown that cycle checking is only then a problem if it is *expensive compared to the execution cost / execution time of operations*. This might be the case for short living transactions, but not in the context of long-running processes in distributed and especially P2P networks that we consider in our approach.

5. SUMMARY AND OUTLOOK

In this paper, we presented the DSGT protocol (Decentralized Serialization Graph Testing), which is designed for decentralized transaction processing in peer-to-peer environments where no central components can be assumed. The protocol distributes the task of coordinating transactions to the set of transactions in the system. In cooperation with the peers they access for executing services, they ensure globally correct executions. Each transaction maintains relevant conflicts in a local serialization graph. Although these graphs usually do not contain full global knowledge, DSGT guarantees serializable schedules. Cyclic dependencies are detected by propagating conflicts along the edges of the local serialization graph like distributed deadlock detection protocols do. These cycles are resolved using a partial rollback approach without losing too much work done by cascading aborts. The experimental evaluation has shown that DSGT significantly outperforms S2PL for medium conflict probabilities and longer transactions composed of expensive services. In case of low conflict probabilities, however, the protocols do not show significant differences. For high conflict probabilities, S2PL is the better choice. Hence, besides the DSGT protocol and the concept of partial rollback for handling isolation failures, this paper has shown that serialization graph testing is well appropriate in service-oriented architectures following a peer-to-peer style of interaction.

In future work, we plan to examine how DSGT can trade freshness of the serialization graph for the quantity of messages. The latter implies to collect changes of the graph and send them in one message instead of immediately propagating each single change.

6. REFERENCES

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

[2] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *VLDB Journal*, 1(2):181–240, 1992.

[3] D. Georgakopoulos, M. Hornick, and F. Manola. Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation. *IEEE TKDE*, 8(4):630–649, 1996.

[4] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data*, pages 173–182, 1996.

[5] K. Haller, H. Schuldt, and C. Türker. A Fully Decentralized Approach to Coordinating Transactional Processes in Peer-to-Peer Environments. Technical Report 463, ETH Zurich, Switzerland, October 2004. `ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/4xx/463.pdf`.

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

[7] N. Krivokapic, A. Kemper, and E. Gudes. Deadlock Detection in Distributed Database Systems: A New Algorithm and a Comparative Performance Analysis. *VLDB Journal*, 8(2):79–100, 1999.

[8] H. Kung and J. Robinson. On optimistic Methods for Concurrency Control. *ACM TODSs*, 6(2), 1981.

[9] E. Levy, H. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. *In Proc. of ACM SIGMOD*, pages 8897, 1991.

[10] R. Obermarck. Distributed Deadlock Detection Algorithm. *ACM TODS*, 7(2):187–208, 1982.

[11] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.

[12] M. Roesler and W. A. Burkhard. Resolution of Deadlocks in Object-Oriented Distributed Systems. *IEEE Transactions on Computing*, pages 1212–1224, August 1989.

[13] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, March 2002.

[14] C. Türker, K. Haller, C. Schuler, and H.-J. Schek. How can we support Grid Transactions? Towards Peer-to-Peer Transaction Processing. In *Proceedings of the Second Conference on Innovative Data Systems Research, CIDR 2005*, pages 174–185, 2005.

[15] 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.

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