A Fully Decentralized Approach to Coordinating Transactional Processes in Peer-to-Peer Environments

Author(s):
Haller, Klaus; Schuldt, Heiko; Türker, Can

Publication Date:
2004-10

Permanent Link:
https://doi.org/10.3929/ethz-a-006775448

Rights / License:
In Copyright - Non-Commercial Use Permitted
A Fully Decentralized Approach to Coordinating Transactional Processes in Peer-to-Peer Environments

Klaus Haller, Heiko Schuldt, Can Türker

Technical Report #463, 2004

October 2004
A Fully Decentralized Approach to Coordinating Transactional Processes in Peer-to-Peer Environments

Klaus Haller\textsuperscript{1} Heiko Schuldt\textsuperscript{2} Can Türker\textsuperscript{1}

\textsuperscript{1} Swiss Federal Institute of Technology Zurich
Institute of Information Systems
CH-8092 Zurich, Switzerland
\{haller,tuerker\}@inf.ethz.ch

\textsuperscript{2} UMIT Innsbruck
Information and Software Engineering
A-6020 Innsbruck, Austria
heiko.schuldt@umit.at

Abstract. With the proliferation of e-business, peer-to-peer style business collaboration becomes increasingly popular. In peer-to-peer environments each peer provides a set of services. These services can be composed to processes running over several 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 novel distributed serialization graph-based approach to 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, up-to-date serialization graph. The protocol thereby fully decentralizes global transaction coordination. 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 sophisticated partial backward recovery is applied. We present in detail the protocol that jointly addresses concurrency control and recovery and provide the proof of its correctness.

1 Introduction

The proliferation of e-business technologies has made service-oriented computing 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 [123].

Workflow and process technologies support this kind of service composition, usually provided by sophisticated system infrastructures like IBM’s WebSphere [4]. However, such systems require a global coordinator (or a set of such coordinators replicated

\footnote{This work is funded in part by the Swiss National Science Foundation within the project MAGIC.}
for performance reasons). While this can easily be enforced for well-established business interactions, it is no longer true when interactions rather follow an ad-hoc style. The increasing trend towards peer-to-peer style collaboration has even reinforced the need for a truly decentralized coordination of process executions.

In peer-to-peer 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 level of processes. This demands for a concurrency control and recovery technique that respects the peer-to-peer style of communication between the system components and that is able to handle large-scale networks of autonomous peers.

Conventionally, isolation and atomicity are enforced using a locking protocol like the strict two-phase locking protocol (2PL) in combination with a global commit protocol like the two-phase commit protocol (2PC) [5,6]. Such protocols are usually applied to short living transactions. However, they are not well suited for peer-to-peer environments since they require a central coordination component. Moreover, they are too restrictive since they in general lead to unnecessarily long blocking of transactions and might cause unnecessary rollbacks.

In this paper, we present a novel approach to concurrency control and recovery that can be applied to peer-to-peer environments. Essentially, it ensures globally correct executions without involving a global coordinator. The main idea of the 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 logs local conflict information 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 [7], 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 it is neither involved in a cycle nor dependent from another transaction, it is allowed to safely commit. Conversely, if a transaction detects a cycle in the serialization graph, the latter has to be resolved by rolling back one or more transactions involved in this cycle. Here, partial backward 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 the system model together with some basic definitions. Section 3 explains how our protocol ensures correctness, whereas Section 4 concentrates on cycle detection in a peer-to-peer environment. Our approach to recovery is the content of Section 5. Finally, Section 6 discusses related work, and Section 7 concludes.
2 System Model

We assume a peer-to-peer network where each peer is able to communicate with all other peers of the network. As illustrated in Figure 1, each peer $P_i$ offers a set of services $O^{P_i} = \{s_1^{P_i}, s_2^{P_i}, \ldots, s_n^{P_i}\}$. The services of a peer can be invoked within transactions using the service interface of that peer.

**Definition 1.** A transaction $T_i$ is a pair $(O_T, \prec_T)$, where $O_T$ is the set of services to be invoked and $\prec_T$ is a total order defined over $O_T$. $^1$

To initiate and execute transactions, each peer of the network provides a transaction execution environment. This environment allows for invoking services on any peer of the network.

Note that a transaction may not always succeed due to several failure reasons. To satisfy the demand for an atomic execution, the transaction must compensate the effects of all the services it has invoked prior to the failure. This compensation is performed by invoking semantically inverse services in reverse order (cf. 3). We therefore rely on the following assumption:

**Assumption 1** For each service $s_j$, the corresponding peer provides an inverse service $s_j^{-1}$ that semantically undoes the effect of the invocation of the original service $s_j$. $^1$

$^1$ The approach proposed in this paper also supports partially ordered service invocations. However, for the sake of clarity but without imposing limitations on the overall approach, we restrict this definition to a total order.
Note that the inverse service might also be the "empty service". The effects of the inverse services strongly depend on the semantics of the original service.

The notion of a schedule is fundamental for defining a criterion for correct concurrent executions of transactions — although no central scheduler exists and thus no complete schedule is materialized in the system.

Definition 2. A schedule $S$ is a pair $(O_S, <_S)$ with $O_S$ being the service invocations and $<_S$ the order between these invocations.

We use the notion of conflict preserving serializability (CPSR)\cite{8} as correctness criterion. According to this criterion, a schedule is correct if and only if there exists a conflict-equivalent serial schedule. A schedule is called serial if the transactions involved in this schedule are entirely executed in sequential order. The serialization graph in which each node corresponds to a transaction of that schedule and where the edges correspond to conflicts between these transaction reflects the dependencies within a schedule. Then, the schedule is conflict-equivalent to a serial schedule if and only if the serialization graph is acyclic\cite{8}. In this case, the schedule is called serializable.

In what follows, the notion of conflict is defined based on the commutativity behavior of service invocations. Dependencies between transactions in a schedule occur when there is at least a pair of service invocations that is in conflict.

Definition 3. Let $s_1$ and $s_2$ be two service invocations. Then, $s_1$ and $s_2$ commute if for any pair of sequences $\alpha, \beta$ of service invocations the results (return values) are the same in $\langle \alpha s_1 s_2 \beta \rangle$ and $\langle \alpha s_2 s_1 \beta \rangle$. Otherwise, the service invocations are in conflict, i.e., $(s_1, s_2) \in \text{CON}$ with $\text{CON} \subseteq \{ O_{T_1} \ldots O_{T_n} \} \times \{ O_{T_1} \ldots O_{T_n} \}$ being the conflict relation.

Definition 4. A transaction $T_i$ depends on a transaction $T_k$ in a schedule $S$ if there exists a pair of conflicting service invocations $s_i \in O_{T_i}$ and $s_k \in O_{T_k}$ such that $s_i$ occurs before $s_k$ in $S$, i.e., $(s_i, s_k) \in \text{CON}$ and $s_i <_S s_k$. This is also denoted by $s_i \rightarrow_S s_k$ and the dependency between the associated transactions as $T_i \rightarrow_S T_k$.

We assume that each peer owns a conflict matrix in which the administrator of a peer defines the conflicts among the services offered by this peer. These conflicts are defined regarding the semantics of the services, according to definition\cite{3}. Using the conflict matrix, a peer is able to detect conflicts between service invocations of different transactions. We however assume that conflicts cannot occur between services of different peers:

Assumption 2. The peers of the network are independent in the sense that conflicts can only appear among service invocations of the same peer. Therefore, service invocations of different peers always commute.

Each peer stores information about the invocation of a service in its local log. Using this information, it can derive conflicts between transactions that invoked services on that peer.

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

The following definition introduces a notion that will be later used for optimizing the graph cycle checking.

**Definition 5.** Let \( SG_T \) be the local serialization graph of transaction \( T \). The nodes that can (transitively) be reached from the node \( T \) via a directed path represent the (transitively) post-ordered transactions of \( T \). Analogously, the nodes for which there is a directed path to the node \( T \) are called the (transitively) pre-ordered transactions of \( T \).

**Definition 6.** The reduced graph \( SG_{T}^{red} \) of \( T \) is derived from \( SG_T \) by eliminating all nodes (together with its corresponding edges) that do not represent a pre- or post-ordered transaction of \( T \).

A criterion for reasoning about the correct concurrency control alone is not sufficient. Additionally, the system must be able to recover from failures.

**Definition 7.** A schedule \( S_i \) is recoverable if and only if the following is true for any pair of transactions \( T_j, T_k \): \( s_x \in T_j, s_y \in T_k, (s_x, s_y) \in CON, s_x <_S s_y \Rightarrow c_j <_S c_k \). \( c_j \) and \( c_k \) denote the commit of transaction \( T_j \) and \( T_k \), respectively.

In a peer-to-peer network, there is no component maintaining the overall knowledge of the system, i.e., the global schedule \( S_G \). Instead, each peer \( P_1, P_2, \ldots \) has a local schedule \( \{ S_{P_1}, S_{P_2}, \ldots \} \), which logs the locally performed service invocations. The global schedule corresponds to the union of the local schedules.

**Definition 8.** Let \( S_L \) be the set of local schedules \( S_{L_i} = (O_{L_i}, <_{L_i}) \), \( 1 \leq i \leq n \). Then, the global schedule \( S_G = (O_G, <_G) \) is constructed as follows: \( O_G = \bigcup O_{L_i} \) and \( <_G = \bigcup <_{L_i} \).

Since we assume that services of different peers always commute, the global schedule cannot contain additional dependencies.

### 3 Ensuring Correctness

Our system model assumes that there is no centralized component with complete global knowledge, e.g., in form of a global serialization graph, which would allow a global coordinator to easily ensure globally serializable schedules. If this was the case, the global coordinator would easily be able to reject or delay the commit of a transaction if this transaction is dependent on another active transaction \([5]\). Consider the graph illustrated in Figure 2. There are four transactions \( (T_A, \ldots, T_D) \) which have already committed and some other transaction \( (T_1, \ldots, T_7) \) that are still active. From the set of active transactions, \( T_1 \) and \( T_7 \) are allowed to commit since they are not dependent on any active transaction. The other ones \( (T_2, \ldots, T_6) \) are not yet allowed to safely commit.
In the following, we present a protocol that is able to enforce globally serializable schedules in a completely distributed way without relying on complete global knowledge. The protocol relies on the observation that by cooperation, transactions and peers are able to enforce that a transaction does not commit if it depends on another active transaction. At commit time of a transaction, it must be ensured that the transaction knows about all conflicts it is involved in. If the transaction depends on any other transaction it must delay its commit until all these transactions have committed. A transaction can get the information about these transactions from the peers on which it has invoked services. At service invocation time, the corresponding peer can determine the local conflicts using its local log. If a conflict occurs, the peer sends the information about the conflict to the transaction together with the result of the service invocation. In this way, each transaction knows exactly about the transactions it depends on.

Algorithm 1 is the part of the protocol that runs on each peer. A 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 the result of the service invocation together with a complete list of conflicts that have occurred to the invoking transaction. This list of conflicts contains all service invocations of other transactions at this peer that are in conflict with the current invocation.
- If a transaction $T_m$ wants to commit, the peer provides a list with all transactions that depend on this committing transaction. This information is needed to inform the dependent transactions about the commit of this transaction (the latter ones might already wait for the commit of $T_m$ in order to commit themselves).

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

The main 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 2).

Then, the peers execute the services as specified in Algorithm 1. Thereby, 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 thread enters the second phase, the validation phase. The transaction now has to wait until it does not depend on any other transaction. As soon as this condition is fulfilled...
Algorithm 1: Sketch of the Peer Protocol

1. while true do
2.   wait for next message $m$;
3.   switch message type of $m$ do
4.     case $T_{invoking}$ invokes service $s_i$ do
5.       execute service $s_i$;
6.       set $T^{conflicting} := \emptyset$;
7.       foreach $e \in \log$ do
8.         if $e.T \neq T_{invoking} \land (e.service, s_i) \in CON$ then
9.             $T^{conflicting} := T^{conflicting} \cup T_{invoking}$;
10.       end
11.       add $(s_i, T_{invoking})$ to Log;
12.     end
13.     case $T_c$ commits do
14.       // collect all dependent transactions
15.       $T^{post} := \emptyset$;
16.       foreach $e \in \log$ with $e.T = T_c$ do
17.         foreach $e' \in \log$ with $e' > e$ do
18.             if $e'.T \neq T_c \land (e.service, e'.service) \in CON$ then
19.                 $T^{post} := T^{post} \cup \{e'.T\}$
20.             end
21.         end
22.       end
23.       // remove all log information of committing transaction $T_c$
24.       foreach $e \in \log$ do
25.         if $e.T = T_c$ then
26.             remove $e$ from Log;
27.         end
28.       end
29.     end
30.     end
31.   end
32. end
33. end

the main thread continues to the commit phase, in which the peers are informed about the commit. Additionally, the transaction informs other transactions which are dependent on it about its commit. This is necessary since these other transactions might wait for this commit in order to safely commit as well subsequently. So, the serialization graph update thread of the other transactions receives this information and changes the corresponding local serialization graph.

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.
**Algorithm 2**: Sketch of the Transaction Protocol

1. $S_T := [s_1, \ldots, s_n]$; // sequence of services to be invoked by $T$
2. $P_T := \{\}$; // peers on which $T$ invoked services
3. $SG_T := \{\}$; // local serialization graph of $T$

4. **Main Thread:**
   5. // 1. Execution phase: invoke services and update serialization graph
      6. **foreach** $s_i \in S_T$ **do**
         7. invoke $s_i$ at an appropriate peer $p$ and add $p$ to $P_T$;
         8. wait for reply from $p$;
         9. update $SG_T$ based on reply information;
         10. propagate changes to pre-ordered transactions $T_s$ (if $SG_T$ is changed);
      **end**
   11. // 2. Validation phase: wait until all pre-ordered transactions have committed
      12. wait until there is no incoming edge to node $T$ in $SG_T$;
   13. // 3. Commit phase: inform all peers such that they can clean up their logs
      14. **foreach** $p \in P_T$ **do**
         15. send ‘commit’ to peer $p$;
         16. update $SG_T$ based on reply information;
      **end**
   17. mark $T$ as ‘committed’ in $SG_T$;
   18. propagate updated $SG_T$ to post-ordered transactions;
   19. **terminate**;

20. **Serialization Graph Update Thread:**
21. **while** true **do**
22. **if** new graph message $SG_{new}$ arrived **then**
23. let $SG_T^{red}$ be the reduced version of $SG_T \cup SG_{new}$;
24. propagate $SG_T^{red}$ to pre-ordered transactions (if $SG_T^{red}$ changed);
25. **end**
26. **if** $SG_T$ changed ∧ $SG_T$ cyclic ∧ $T$ is victim **then**
27. abort;
28. **end**
29. **end**

and is therefore allowed to commit. This is closely related to well-established optimistic concurrency control protocols like backward-oriented concurrency control, BOCC [9] and to serialization graph testing protocols [10]. As in all other cases, transactions are allowed to commit in our approach only if they do not depend on active (uncommitted) transaction. Transactions get this information from the peers they invoke services on.

We can proof the following property of our protocol.

**Theorem 1.** Algorithms 1 and 2 together generate only schedules for which the following holds: No transaction commits before all transactions it is dependent on have committed.
Proof. Assume a global schedule \( S \) in which a transaction \( T_c \) has committed although it is dependent on at least one active transaction. Without loss of generality, then there must be a transaction \( T_p \) such that the conflict \( T_p \rightarrow_S T_c \) holds.

If \( T_c \) has committed, it must have successfully passed the validation in line 13. In this case, the serialization graph of \( T_c \) cannot have contained a conflict leading to the edge \( T_p \rightarrow_S T_c \) at that time. There are two possibilities when this can happen:

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

Both cases however lead to a contradiction to our assumption.

1. Since the edge \( T_p \rightarrow_S T_c \) ends at node \( T_c \), \( T_p \) appeared as a consequence of a service invocation on a peer sent by \( T_c \). This peer would have returned \( T_p \rightarrow_S T_c \) and \( T_c \) inserted it in its graph (lines 8, 9). Hence, \( T_c \) would have known about the conflict \( T_p \rightarrow_S T_c \) in line 13. Consequently, \( T_c \) would not have proceeded and therefore not committed. This however contradicts our assumption.

2. The removal (invalidation) of the conflict \( T_p \rightarrow_S T_c \) with the active transaction \( T_p \) can theoretically only happen in the following situations:
   (a) \( T_c \) marks \( T_p \) as committed. This only appears as a consequence of \( T_p \) marking itself as committed (line 14) and spreading this information. Since we assumed that \( T_p \) is still active, this cannot be the case.
   (b) The edge \( T_p \rightarrow_S T_c \) has been removed due to a wrong ‘optimization’ (line 25).

   However, this is impossible since \( T_p \rightarrow_S T_c \) ends at \( T_c \).

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

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

An important aspect of the commit phase is that transactions willing to commit eventually succeed when all pre-ordered transactions have committed. 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.
Each peer checks for relevant conflicts \((t = t_c + 1)\) and return this information to the issuing transaction \((T_i)\) and remove the entries of the committing transaction from the log \((t_c = t + 2)\).

**Theorem 2.** If a transaction has executed all its service invocations and all the transactions it depends on have committed, then the transaction eventually commits.

**Proof.** Assume \(T_c\) is a transaction which is in the validation phase. Without loss of generality, we assume that \(T_p\) is an already committed transaction and that \(T_p \rightarrow S T_c\) appeared on \(P_p\). Suppose furthermore that \(T_c\) does not get the information about the invalidation of the conflict.

1. \(T_p\) committed (line 21) after it has marked itself as having committed (line 19) and propagated this information (line 20). If \(T_c\) does not eventually get the conflict \(T_p \rightarrow S T_c\), \(T_p\) cannot have known about this conflict, too. Otherwise, it would have informed \(T_c\). In the following, this will be referred to as (*)

2. \(T_p\) contacted \(P_p\) before its commit. \(P_p\) therefore returned all transactions which were dependent on \(T_p\) at this point in time. There are two possibilities:

   (a) The conflict \(T_p \rightarrow S T_c\) did not exist at that time. Then, \(P_p\) removed the log entries related to \(T_p\). Therefore, \(P_p\) could not have returned the conflict \(T_p \rightarrow S T_c\) to \(T_c\). This however is a contradiction to (*).

   (b) The conflict \(T_p \rightarrow S T_c\) existed and was hence returned by \(P_p\) as a reply to the commit information of \(T_p\). In consequence, \(T_p\) must have known about it. This also contradicts to (*).

Thus, our initial assumption cannot hold which proves our theorem.

\[\square\]

## 4 Cycle Detection

A transaction involved in a cycle must detect this, which requires that it receives the relevant conflict information. This section explains how transactions have to exchange their local knowledge with other transactions. Obviously, synchronous updates of the local serialization graphs are not appropriate for any kind of distributed environment due to performance reasons [7], so the update is performed in a lazy manner. In consequence, the serialization graphs will not necessarily be always up-to-date.

In case of a cycle in the serialization graph, the following characterize the situation:

1. None of the involved transactions can commit, because there is a cyclic waiting situation.
2. This situation will not change without any intervention.
3. Cycles might be caused by conflicts of more than two transactions that are executed on different peers. So neither a peer nor a transaction is able to detect such cycles relying only on their local knowledge.

The lines 10, 26, and 28-30 (Algorithm 2) implement the information exchange for detecting cycles. The implementations covers three aspects: (i) If a transaction causes
a new conflict, it not only updates its local graph, but propagates the graph to its pre-ordered transactions. (ii) A transaction uses a graph received from another transaction to update its own serialization graph. If this leads to changes, it propagates its updated graph to its pre-ordered transactions. (iii) If the transaction detects a cycle and a victim selection strategy selects itself as the victim, it aborts.

Obviously, this approach is heavily inspired by distributed deadlock detection algorithms, especially by path pushing approaches (e.g., [11]).

Figure 4 illustrates the propagation mechanism. It is based on a system with four active transactions. The figure shows the local graphs of $T_1$ and $T_3$. At $t = 1$, $T_1$ 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 on to $T_4$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4.png}
\caption{Cycle Detection}
\end{figure}

Based on the implicit assumptions that messages are eventually delivered and the number of transactions is finite, we can proof that the cycle detection works correctly.

**Theorem 3.** Algorithm 1 and 2 ensure that each transaction (eventually) detects a cycle it is involved in.

**Proof.** Let $T_1, \ldots, T_n$ be involved in a cycle $T_1 \rightarrow T_2 \rightarrow \ldots T_i \rightarrow T_{i+1} \rightarrow \ldots \rightarrow T_n \rightarrow T_1$. Thus, proving the theorem requires to proof that each of these transactions gets the information of each $T_i \rightarrow T_{i+1}$ with $n + 1 \equiv 1$.

Without loss of generality, we show that $T_i$ receives the information about each conflict $T_i \rightarrow T_{i+1}$. We prove this in two steps:

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

We start with statement (1):

1. A service invocation of $T_{i+1}$ causes the conflict $T_i \rightarrow T_{i+1}$ by invoking a service on a peer $P_k$. Thus, $P_k$ returns this conflict to $T_{i+1}$. $T_{i+1}$ inserts this conflict into
its graph (Algorithm 2, lines 8–9). Afterwards, $T_{i+1}$ sends this information to $T_i$ (line 10). Therefore, statement (1) holds.

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

(a) $SG_{T_{i+1}}$ already contains $T_i \rightarrow T_{i+1}$, when the message arrives (line 24). Then, $T_{i+1}$ inserts the conflict $T_i \rightarrow T_{i+1}$ into its local graph (line 25). Afterwards, $T_{i+1}$ sends its updated graph to its pre-ordered transactions (line 26). Since $T_i$ is the recipient of this message, this information finally arrives there.

(b) $SG_{T_{i+1}}$ does not contain the conflict $T_i \rightarrow T_{i+1}$. When the message arrives (line 24), $T_{i+1}$ inserts this conflict into its local graph (line 25). Because $T_i$ is not known to be pre-ordered with respect to $T_{i+1}$, $T_i$ is not a receiver when $T_{i+1}$ forwards the updated graph (line 26).

However, according to protocol, $T_{i+1}$ will receive the conflict $T_i \rightarrow T_{i+1}$ from the corresponding peer. Then, $T_{i+1}$ inserts this conflict into the local graph (lines 8–9). Now $T_i$ is a pre-ordered transaction from the point of view of $T_{i+1}$. Therefore, the graph which already contains $T_j \rightarrow T_{j+1}$ is sent to $T_i$ (line 10).

Thus, also $T_i$ receives the information on this conflict. Since cases (a) and (b) are supported, statement (2) also holds such that cycles are eventually detected by the associated transactions.

\[ \square \]

5 Partial Rollback

Failure handling in process and workflow management usually applies advanced concepts like alternative executions [12] rather than supporting the all-or-nothing semantics of atomicity. By this, failures can be handled at the application level (e.g., by executing alternative activities that aim at making a train reservation in case a flight cannot be booked). However, the semantic approach by considering alternative execution paths does not apply to handle failures occurring in concurrent executions when processes access shared resources (isolation failures). The latter should rather be treated transparently and shielded from the application programmer. In addition, also these failures should not lead to the complete compensation of all the effects of an active process. Therefore, we propose partial rollback as a technique for dealing with executions that are not serializable (when a cycle in the serialization graph has been detected). Partial rollback means to rollback a transaction by invoking inverse services only to a point at which the cycle has disappeared. Then, process execution is continued. Usually, partial backward recovery cannot be realized by considering only one process in isolation. Rather, several processes are involved in failure handling. Once the cycle in the serialization graph has been eliminated, it has to be made sure that the cycle does not appear again when all transactions are re-started.

Partial backward recovery is especially useful for addressing the problem of cascading aborts, since it restricts the necessary aborts. Now only one “victim” is aborted completely in case of a cycle whereas the others only rollback as far as it is really necessary. So in Figure 2, $T_4 \ldots T_6$ do not have to rollback completely (as it is the case when applying complete rollback). It is sufficient to abort $T_4$, while all other transactions compensate only a few steps before they restart again. Choosing the “proper” victim by using graph information allows to improve this approach further.
In the following, we formalize our approach to partial backward recovery. Regarding the correctness criterion, we follow the ideas of the unified theory of concurrency control and recovery (UT) \[13\]. This theory replaces aborts by a sequence of inverse service invocations in reverse order of the associated forward service invocations, followed by a commit. However, and in contrast to the UT, we not only support complete rollbacks. Moreover, partial recovery can be applied several times to a process, if needed, i.e., it might execute in a “swinging” way by switching from forward to backward execution and vice versa (although it has to be made sure that the same cycle will not be created again after a restart). For analyzing the correctness of partial backward recovery, we introduce a reduced schedule, which is constructed out of a regular one with the help of the following reduction rules also known from the UT.

**Definition 9. Undo Commutativity Rule.** If \( s_1 \) and \( s_2 \) are two service invocations from different transactions such that \( s_1 < s_2 \) and the pair \((s_1, s_2)\) does commute, and there is no \( p \) such that \( s_1 < p < s_2 \), then the order \( s_1 < s_2 \) can be replaced by \( s_2 < s_1 \).

**Definition 10. Undo Reduction Rule.** Let \( s \) and \( s^{-1} \) be two service invocations with \( s < s^{-1} \) and there is no \( p \) with \( s < p < s^{-1} \). Then, \( s \) and \( s^{-1} \) can be safely removed from the schedule.

This allows to formulate the following correctness criterion, which reduces the schedules to ones for which the CPSR criterion is applicable:

**Definition 11. Correct Execution with Partial Rollback (CEPR)** Let \( S \) be a schedule that reflects a concurrent process execution where partial rollback might have been applied. Then, \( S \) is correct iff a serial schedule \( S' \) can be constructed out of \( S \) by applying the undo commutativity rule and the undo reduction rule in arbitrary order and in arbitrary number.

Enforcing the CEPR criterion by implementing a protocol is straightforward. Processes invoke inverse services in reverse order \[13\]. For example let
\[
s_a^2, s_e^2, s_f^2, s_d^2, s_b^1, s_a^1
\]
be a schedule, with \( s_1^1 \) representing services invoked by \( T_1 \) and \( s_2^2 \) services invoked by \( T_2 \). All services are executed on the same peer. Furthermore, let \( s_a^2 <_S s_b^1 \) and \( s_b^1 <_S s_b^2 \) be the conflicts such that the graph is cyclic. In case of complete rollback, all services would have to be compensated. However, assuming that \( T_1 \) is chosen as victim, it is sufficient to compensate only three service invocations \((s_a^1, s_f^1, s_a^1)\). Then, the cycle is removed. \( T_2 \) executes again \( s_b^2 \) and \( s_a^2 \) whereas \( T_1 \) restarts from the beginning. This is achieved by our protocol as follows: \( T_1 \) contacts the peer to compensate \( s_a^1 \), which is done immediately by the peer (note that we did not include partial rollback in the algorithms presented in this paper; however, more details on this can be found in \[14\]). Then, \( T_1 \) wants to compensate \( s_b^1 \). This time, the peer realizes that \( s_b^2 \) is in conflict with \( s_b^1 \). Therefore, \( s_b^2 \) must be compensated first. So \( T_1 \) contacts \( T_2 \) and requires it to compensate \( s_b^2 \). This can be done by \( T_2 \) immediately. Afterwards, \( T_2 \) informs \( T_1 \) about that such that \( T_1 \) tries again to compensate \( s_b^1 \). This time, \( T_1 \) succeeds.
6 Related Work

Transaction processing in peer-to-peer systems was neglected by previous research. In principle, protocols that have been developed for distributed and federated database systems are applicable, but have minor and also severe drawbacks. In the following, we briefly review the various directions of protocols. Optimistic protocols, such as proposed in [9], execute transactions without any validation. At commit time, it is checked whether the execution of a transaction was correct with respect to the isolation criterion, i.e., whether the produced schedule is serializable. Optimistic approaches 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, such as proposed in [13], however, stick to a global coordinator, which makes them of limited use for peer-to-peer environments.

Pessimistic protocols prevent schedules from becoming temporary incorrect by checking prior to the execution of each operation whether or not it leads to incorrect schedules. Strict two-phase locking (S2PL; [5]) is a well-known pessimistic approach that allows an efficient implementation and is well suited for short transactions. However, holding locks for the whole execution time of the transaction implies that transactions easily block each other. This results in poor overall throughput if the transactions are long-running. Nevertheless, in principle, S2PL can be used in distributed environments together with a distributed deadlock detection protocol, such as proposed in [11] or [16]. An interesting approach that runs a distributed deadlock detection agent on each partition of the serialization graph is presented in [17]. This idea inspired us to distribute the serialization graph in fully decentralized manner.

Timestamp ordering protocols are interesting in distributed environments since they do not require any coordination of different resources [5]. Instead, timestamp ordering attaches a unique timestamp to each transaction, by which a total order over all transactions is established. The execution order of conflicting operations must stick to this order. Otherwise, a transaction has to abort. Thus, the longer transactions are and the more transactions execute in parallel, the more likely it is that this total order is violated (though the schedule is correct) requiring unnecessary aborts. The ticket method [18] provides a timestamp-based solution for federated database systems where indirect conflicts are made explicit (globally known) via ticketing. The applicability of such kind of approaches is restricted in peer-to-peer environments since we cannot expect to have such a global ticketing component.

Moreover, there are serialization graph tester which map transactions to nodes and dependencies between transactions to directed edges of a serialization graph. A schedule is serializable if and only if the corresponding serialization graph is acyclic [5]. A coordinator maintains this graph and checks before the execution of an operation whether a cycle would arise due to the execution of this operation. If so, the coordinator has to chose one or more transactions as victims for abortion. The serialization graph approach can also be used in a distributed environment where the local serialization graphs can be combined to a global one [19].

Due to the cycle checking, serialization graph testing has a linear run-time complexity. 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
Coordinating Transactional Processes in Peer-to-Peer Environments

as a formal method to explain serializability theory. Interestingly, previous research has not seen that cycle checking is only then a problem if it is expensive compared to the operation execution cost. This might be the case for short living transactions, but not in the context of long-running processes in distributed and especially peer-to-peer networks.

7 Summary and Outlook

In this paper, we presented a fully decentralized approach to transaction management in peer-to-peer environments. The proposed approach is well-suited especially for long-running transactional processes. Our protocol is based on distributed serialization graph testing. Each transaction owns a local serialization graph that reflects the conflicts its is involved in. Based on its not necessarily up-to-date serialization graph, each transaction is able to decide on its own whether it is allowed to commit without violating the global correctness criterion of conflict-serializability.

Cyclic waiting situations are detected by the involved transactions using a path-pushing approach that is similar to that ones used for distributed deadlock detection. In our approach, each transaction propagates only the “relevant” edges of its serialization graph to the transactions on which its depends. Relevant edges are those ones that might cause a cycle in which the corresponding transaction would be involved in.

To resolve cycles, we propose a partial rollback approach. The victim is of course compensated completely. The other transactions involved in the cycle partially rollback as far as needed for compensating the victim completely. Then, these transactions restart from point. In this way, for many situations the problem of cascading aborts can be reduced significantly. This is a very important issue for long-running transactions since otherwise cascading aborts would imply to throw away huge amount of work done until the failure situation emerged.

Extensive performance measurements of our protocol in a distributed application server setting — which were out of the scope of this paper — show highly convincing results. Our protocol clearly outperforms locking-based protocols in case of (long-running) transactions in peer-to-peer environments although it of course still requires (linear-time) graph cycle checking. The interesting point here is that the time needed for cycle checking is relatively small with respect to the duration of long-running transactions.

In future work, we want to figure out in detail the influence of victim selection and how we 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 propagating each change immediately.

Acknowledgements. We would like to thank Hans-Jörg Schek and Yuri Breitbart for their valuable discussions.

References