Transactions Concurrency Control in Web Service Environment

Mohammad Alrifai
L3S and University of Hanover,
Expo Plaza 1, 30539 Hanover, Germany
alrifai@l3s.de

Peter Dolog
Aalborg University,
Department of Computer Science,
Fredrik Bajers Vej 7, DK-9220 Aalborg Ø
dolog@cs.aau.dk

Wolfgang Nejdl
L3S and University of Hanover,
Expo Plaza 1, 30539 Hanover, Germany
nejdl@l3s.de

Abstract

Business transactions in web service environments run with relaxed isolation and atomicity property. In such environments, transactions can commit and roll back independently on each other. Transaction management has to reflect this issue and address the problems which result for example from concurrent access to web service resources and data. In this paper we propose an extension to the WS-Transaction Protocol which ensures the consistency of the data when independent business transactions access the data concurrently under the relaxed transaction properties. Our extension is based on transaction dependency graphs maintained at the service provider side. We have implemented such a protocol on top of WS-Transaction. The extension on the web service provider side is simple to achieve as it can be an integral part of the service invocation mechanism. It has also an advantage from an engineering point of view as it does not change the way consumers or clients of web services have to be programmed. Furthermore, it avoids direct communication between transaction coordinators which preserves security by keeping the information about business transactions restricted to the coordinators which are responsible for them.

1. Introduction

Web services are distributed independent computing units that provide a Business-to-Business interface based on standards like SOAP [8] and WSDL [9]. They allow the integration and collaboration of different business applications running on different heterogeneous back-end systems. BPEL4WS [10] is a workflow-like definition language that describes sophisticated business processes that can orchestrate web services. Most of the web service-based business processes are long-running transactional processes. Traditional protocols like the strict two-phase locking protocol (2PL) [12] and the two-phase commit protocol (2PC) [13], which are based on resources locking mechanisms, are impractical in such environments. For instance, a web service provider would not accept to lock its local resources for a long time by the web service consumers. Thus, traditional ACID (Atomicity, Consistency, Isolation and Durability) properties of a transaction management system have to be relaxed for web services-based transactions. In particular, atomicity and isolation properties are usually relaxed in existing proposals for transaction protocols in the web service environment; i.e. some activities in a transaction can commit their results before the whole transaction commits and the results of some activities can be seen before the whole transaction completes. However, several problems arise because of the relaxation of the ACID properties such that a readdressing of the transaction management problem for web services environment is required.

Figure 1 shows a conceptual view of the problem. Several transactional business processes (T_1 and T_2) concurrently invoke web services (WS_i) from different providers (P_1 and P_2). Transactional support is required in order to reach a mutually agreed upon outcome for the whole web services within one transaction from the client's viewpoint, as well as to ensure that the outcome of all transactions are consistent from the provider's viewpoint.

A web service, which has already completed its task and returned the outcome to its invoking business process, may later need to undo its job in case of the failure of another web service within the same transaction. Therefore, support for recovery

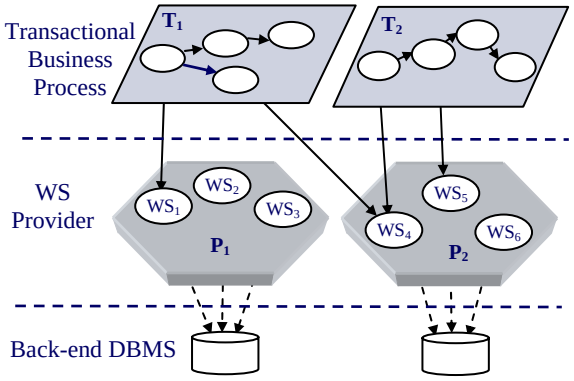

Figure 1. Conceptual View over the Transaction Problems in Web Service Environment

mechanisms is required for web services. And because of the Isolation-relaxation property, completion dependency relations occur between concurrent transactions that access shared resources, therefore, a dependent transaction should not commit before its preceding one does. To handle such dependencies, support for a concurrency control mechanism is needed.

Previous proposals for transactional management support in web service environment either address one part of the problem, the recovery problem [1, 2, 3, 4 and 5] or they involve several independent transaction coordinators communicating with each other [6, 7] to achieve the concurrency control.

While these approaches are important first steps, they neglect that by sharing the dependencies information possibly mission-critical information is disclosed to parties which should not have access to them. In these cases, direct communication between transactions should definitively be avoided, which of course raises the question, how we can achieve globally correct execution without communication among transactions.

In this paper we therefore propose a protocol for addressing the concurrency control problem in a way that does not require any direct communication between transactions as an extension to the WS-Transaction and WS-Coordination Protocols. The protocol has the following advantages:

- Transaction dependencies are maintained at the service provider side avoiding direct communication between third party transactions.
- As dependent transactions will be informed (by the common participant) as soon as the relevant web service commits or rolls back, they do not have to wait until the dominant transaction either completely commits or completely aborts.

The rest of the paper is structured as follows: Section 2 reviews some related work and in Section 3 we describe a motivating scenario for the problem. In Section 4 we discuss the WS-Coordination and WS-Transaction specifications and their shortcoming for addressing this problem. In Section 5 we define the concurrency control problem for web services and in Section 6 we describe our approach to provide concurrency control for web services by building on top of the existing standards. In Section 7 we discuss our proposed solution and in Section 8 we describe our prototyping implementation to validate the proposed protocol. Section 9 concludes this paper and gives an outlook to future work.

2. Related Work

Existing solutions (WS-Transaction/WS-Coordination [1, 2, and 3], BTP [4] and WS-CAF [5]) address the recovery problem by supporting the coordination of the so-called distributed open-nested transactions. An open nested transaction is a tree (of arbitrary height) of “sub-transactions”. The sub-transactions may commit independently of each other without having to wait for the root transaction to commit. This relaxes the isolation part of the ACID properties. In case of a sub-transaction failure, the client who is driving this business process may decide whether the overall transaction should abort or simply ignore the failed sub-transaction. For example, an ordering system that chooses the cheapest supplier might still be able to commit successfully if only one of the suppliers fails during the transaction (atomicity relaxation). Typically, sub-transactions are matched to the transactions already supported by Web services (e.g., transactional booking offered by a service).

The problem of concurrency control was addressed by [6] and [7]. In [6] the proposed solution is made as an extension to the WS-Transaction Protocol, whereas [7] proposes a completely new protocol based on a decentralized serialization graph test protocol. Both solutions share the same idea: globally correct execution is achieved by direct communication among coordinators of dependent transactions. We argue that such direct communication between transactions should be avoided, as the exchanged dependency information can be interpreted as mission-critical information such as confidential contracts between organizations.

3. Motivating Scenario

An airline company offers 10 tickets to a travel agency to sell to its customers. This offer is restricted to a certain type of airplane. A customer is requesting several flight tickets through a tourist agency. This creates a business transaction which consists of request, selection, confirmation, payment and getting the ticket. The agency requests those seats from an airline company by contacting its reservation service. At the time of the request, the tickets are available. The agency provides the tickets to the customer. The user selects them and tries to book them.

In the meantime, the offer from the airline company cannot be fulfilled as another third party transaction made changes in the flight offerings. Therefore, the user request cannot be committed and has to be rolled back or compensated. This scenario includes a dependency between 2 independent business transactions. The dependencies between such transactions occur dynamically and independently from running services. They are long running processes. Locking would not be therefore acceptable for the company businesses. In long run, many parties can join such business transaction dependencies, where each transaction is coordinated by independent coordinators. So from practical and privacy point of view it would be more desirable to avoid a communication between the transaction coordinators to resolve these dependencies. For instance in our example scenario, it is not the duty of the (business transaction running by the) customer to contact the transactional business process of the airways company to resolve the dependency relation between them because of their concurrent access to the resources held by the travel agency. Instead, such communication can be introduced at the service provider level (the travel agency) where providers maintain the dependency data as inherent part of the web service invocation mechanism which is not supported by current web service transactional management protocols standards.

4. WS-Coordination and WS-Transaction Protocols

Unlike the OASIS Business Transaction Protocol (BTP) [4], which is aimed to representing and seamlessly managing complex business-to-business transactions over the Internet, both WS-Coordination [3] and WS-Transaction [1, 2] specifications (from IBM, Microsoft, BEA and others in the industry) are intended solely for the Web services environment and as such leverage existing and evolving standards, such as WSDL, WS- Addressing, Web Services Security, and WS-Policy.

4.1. WS-Coordination

The WS-Coordination specification [3] defines a framework that is aimed at reaching an agreement on the final outcome of a web services-based transactional process, regardless of the specific transaction protocol being used for this purpose. It defines two key concepts: 1) the Coordinator, which is an entity that resides on the client side and is responsible for reaching a globally agreed upon outcome of the transaction from the client's point of view, 2) the Participant, which is an entity that resides on the web service provider side and is responsible for communicating with the Coordinator according to the protocol on behalf of the web service (Figure 2).

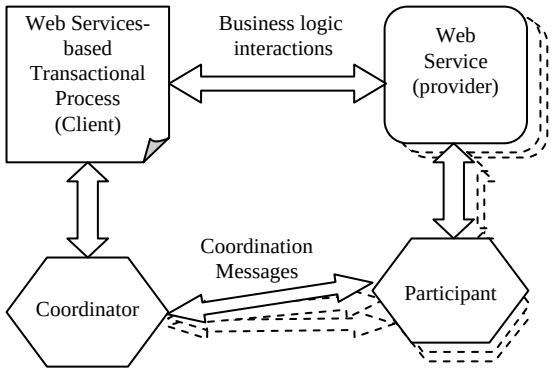

Figure 2. WS-Coordination Architecture

4.2. WS-Transaction

The WS-Transaction specification [1, 2] plugs into the WS-Coordination and describes two transaction protocol models to support the semantics of two kinds of business-to-business interaction: AtomicTransaction (AT), which is similar to the traditional ACID transactions and intended for short-duration interactions, and BusinessActivity (BA), which is intended for long-duration, ACID-relaxed transactions among loosely-coupled systems where exclusively locking resources is impossible or impractical [1, 2].

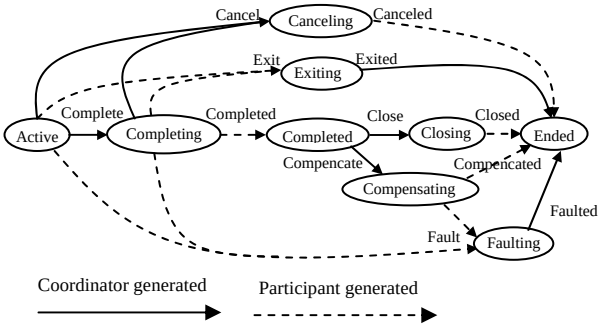

Figure 3. Abstract State Diagram of `BusinessAgreementWithCoordinatorCompletion`

Under the scope of an AtomicTransaction, the coordinator directs all participants either to all commit or all cancel, whereas under the scope of a BusinessActivity, the director may direct each participant individually. In this paper we consider the BusinessActivity model where parallel transactions can have concurrent access to local resources of a given provider through its web services and the need for concurrency control arises because of the Isolation-relaxation property of this model.

4.3. WS-BusinessActivity

This specification defines two specific agreement coordination protocols for the BusinessActivity transaction model that can be used with the extensible coordination framework WS-Coordination: `BusinessAgreementWithCoordinatorCompletion`, and `BusinessAgreementWithParticipantCompletion`. In the former one, the participants rely on the coordinator to inform them when they have received all requests to perform work within the business activity, whereas in the latter one, the participants themselves know when they has completed all requests and should inform the coordinator about that. Figure 3 shows the abstract state diagram for a participant of the `BusinessAgreementWithCoordinatorCompletion` protocol. Upon the receipt of the complete message, the participant knows that it will not receive any new requests or tasks from this transaction so it finalizes its work and makes the results in a way such that it can later either be durably stored or compensated based on the next command received from the coordinator: close (commit) or compensate.

From the provider point of view, the participant may be in conflict with another one from another transaction, i.e. it uses some resource that has been previously updated by another (dominant) participant within different transaction. In such case, the final outcome of the dependent participant depends on the final outcome of the dominant one. Therefore, the dependent participant should not close (commit) before the dominant one does, and should inform its coordinator as a reply to the complete message it has received so that it waits until the dominant participant reaches its final state. And in case that the dominant participant fails or compensates, the dependent one must compensate its work and inform its coordinator about this. The current specifications does not support such functionalities at all, therefore there is a need to extend the specifications to provide a mechanism for supporting the concurrency control.

5. Transaction Completion Dependencies

Our protocol is a variant of the Distributed Serialization (Conflict) Graph Testing protocol [11] to describe the directed graph, which contains all dependencies that can be seen from the provider's point of view. The protocol operates on transaction schedules, which are processed by the providers. A transaction *schedule* is a set of transactions in partial order where some of the transactions may be executed concurrently. Two operations O_1 and O_2 are in *conflict* in a transaction schedule if they are invoked by two different transactions T_1 and T_2 respectively, and access the same resource held by a common provider and at least one of them is influencing the result of the other operation (for example by writing a data item used to compute the output of the other one). We say that there is a *completion dependency* relation between T_1 and T_2, *i.e.* the outcome of the *dependent* transaction is based on the outcome of the *dominant* one.

A *serialization graph* is a directed graph consisting of transactions as nodes and edges representing completion dependencies between them such that the edge points from the dependent transaction to the dominant one.

To maintain consistency, a dependent transaction must delay its completion until all its dominant transactions complete (either commit or abort). In case that a dominant transaction aborts (i.e. the relevant Web Service rolls back), a dependent transaction needs to roll back its affected Web Service. This approach ensures that the concurrent transactions at the end will have consistent outcomes. On the other hand, in case of a cyclic serialization graph this may lead to having some transactions waiting for each other forever. Therefore it is necessary to take this case into account and any transaction management system for web services should be able to handle the following two key issues: How can circular waiting be detected; and if detected, how can it be resolved?

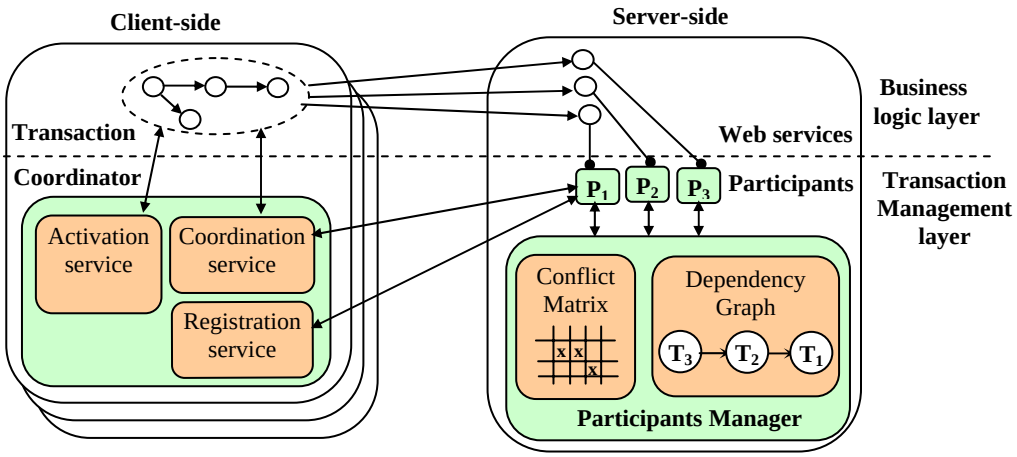

Figure 4. Components of a Transaction-aware Web Service Environment

6. A Support for Concurrency Control for Web Services

6.1. Transaction Dependencies Management

The transaction aware web service environment consists of a set of service providers and a set of service consumers. Each service provider provides one or more operations as web services. A client may initiate transactional processes with web services residing on several servers. When a transaction is initiated it is assigned to a coordinator according to the WS-Coordination specification [3], which ensures that a final mutually agreed-upon outcome will be reached by all the involved web services.

Due to relaxed atomicity and isolation properties, operations from different transactions can run into conflicts. Each service provider maintains information about all completion dependencies between the concurrent transactions and information about the state of each web service with respect to the completion of its invoking transaction. Service providers ensure that all clients will reach a final globally correct state.

6.2. Participants Manager

We introduce the concept of a Participants Manager: an entity which resides on the web services provider side and is responsible for managing the participants of the different concurrent transactions so that it ensures a consistent outcome for all of them. The Participants Manager maintains the conflict matrix, which is built at design time and provided to the Participants Manager as an input to be used to detect any dependency relation between the concurrent transactions at run time. The conflict matrix is a N x N matrix, where N is the total number of the web service operations that can be accessed via the provider. There is a conflict between two operations O_i and O_j if the field F_{ij} in the conflict matrix is set to 1. So, a dependency relation between two concurrent (and still not committed) transactions T_1 and T_2 is detected if T_1 invokes O_i after T_2 has invoked O_j.

The Participants Manager maintains the dependency graph based on the information it gets from the participants that are involved in the running transactions. After each operation invocation, the participant, which represents the invoked operation, informs the Participants Manager about this action. The Participants Manager in turn, checks the conflict matrix and updates the dependency graph accordingly. Figure 4 depicts the components of a transaction aware web service environment.

Whenever a participant receives the complete message from its coordinator, it firstly asks the Participants Manager whether there is any dependency relation with other transactions. According to the response it receives from the Participants Manager, it responds to the coordinator either by a completed or a wait message. If a participant receives a close/compensate message (as a response to a previously sent completed message) it commits/compensates its job and informs the Participants Manager. The Participants Manager in turn removes the corresponding transaction from the dependency graph and informs all the participants of the dependent transactions so that any waiting participant can either safely commit or compensate and inform its coordinator about its final outcome (by sending either completed or compensated message respectively).

Algorithm 1 Service Provider

```
SG = {} (local serialization graph)
O  = {o1,…,oi} (operations at provider)
M  = Conflict Matrix
(body of the service provider program)
activate(oi);
check M and addToSerializationGraph(ti, oi);
wait for next message from coordinator of  ti ;
When message m from coordinator received
switch message type of m do
   case m is cancel
        roll back;
        deleteFromSerializationGraph(ti, oi);
        send message CANCELED
   case m is complete
        complete(); // see algorithm 2
   case m is close
        commit();
        deleteFromSerializationGraph(ti, oi);
        send message CLOSED;
   case m is compensate
        compansate();
        send message COMPENSATED;
   case m is exited
        deleteFromSerializationGraph(ti, oi);
        END;
   case m is faulted
        roll back;
        deleteFromSerializationGraph(ti, oi);
        END;
end switch
Exception When Fault
    send message FAULT;
```

6.3. The Protocol

A dependency is detected when a transaction invokes an operation which is in conflict with operations invoked by already running transactions. The dependency information is stored at the service provider side. Algorithm 1 describes the main execution thread at the provider side. The coordinator of the dependent transaction is informed about the completion dependency by sending a *wait message*. Algorithm 2 describes how the completing phase is performed at the provider side. If a *close* message is received from the dominant transaction, it is propagated to the dependent transaction. If a *compensate* message is received, the operation is compensated and the provider informs all dependent transactions about this.

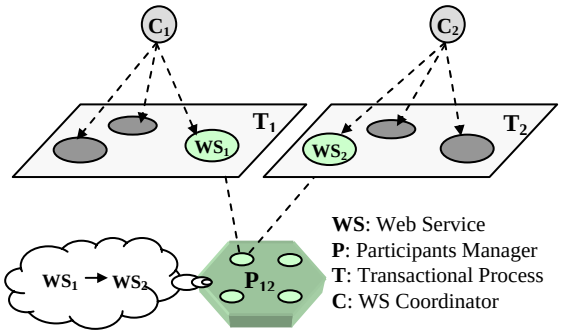

Figure 5. Two Dependent Transactions

Figure 5 depicts two transaction coordinators coordinating two independent transactional processes. The dependencies are detected between T_1 and T_2 based on WS_1 and WS_2. P_{12} detects a dependency relation between WS_1 from T_1 and WS_2 from T_2. P_{12} informs the transaction coordinator (C_1) of the dependant transaction T_1 that it is waiting by sending a waiting message. The coordinator (C_1) of the dependant transaction (T_1) now knows that it should wait for the Web Service in conflict (WS_1), which is held by P_{12} and can inform other participants of T_1 about this if needed. P_{12} waits until it receives a finishing message from the coordinator C_2 of the dominant transaction T_2: either close message for committing or compensate. If P_{12} receives a close message from T_2 (for WS_2), it informs T_1 by sending closed message (for WS_1). If P_{12} receives a compensate* message from T_2 (for WS_2), it compensates WS_2, and then compensates WS_1 and informs T_1 by sending a compensated message (therefore a new transition from the newly created state waiting to the compensating state is needed to be added into the state diagram).

Figure 6 depicts a state diagram with possible web service state and transitions generated either by a transaction coordinator or service provider.

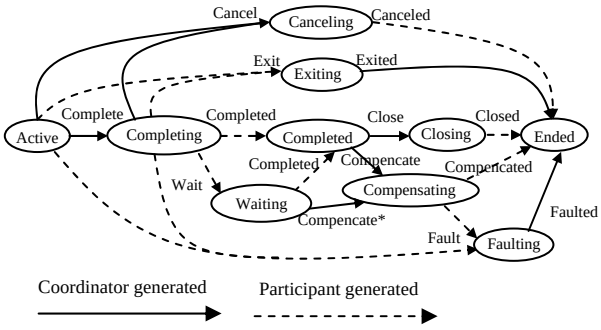

Figure 6. Abstract State Diagram of `BusinessAgreementWithCoordinatorCompletion`

6.4. Cycle Detection

As shown on Figure 7.a the three transactional processes are running into conflict. The dependencies are detected between T_1 and T_2 based on WS_1 and WS_2, T_2 and T_3 based on WS_3 and WS_4, and between T_1 and T_3 based on WS_5 and WS_6. In the worst case, the dependencies will form a cycle, which cannot be seen locally by the providers. Let's assume that T_2 is a dominant transaction of T_1 and T3 is a dominant of T2 and T1 is a dominant transaction of T3. If T1 starts its completion phase at some point of the time, its Coordinator C1 will send a `complete` message to all of its participants including the (participant of) WS_1. According to our protocol, WS_1 will get the dependency information from the Participants Manager P12 and will send a wait message to C1. The same holds for T2 and T3, hence it will turn out that all coordinators will run into a waiting cycle since C2 is waiting for C3 and C3 is waiting for C1, which is waiting for C2.

To solve this problem, we need a mechanism to detect such cycles and to resolve them, so that all involved transactions can safely complete. A naive approach would be to inform the coordinators of each transaction about the cycle and to make to make them communicate with each other to detect the cycle as soon as possible and start resolving it. However, we think that direct communication between transactions should be avoided. The exchanged dependency information can be interpreted as mission-critical information such as confidential contracts between organizations and therefore not to be provided to other independent transactions of third parties. Therefore we propose another mechanism that solves the problem without such kind of communications between the independent transactions. When a participant is in a waiting state it starts a so-called WaitingCycleCheck procedure to detect any potential cycles. In waiting state, the provider asks coordinators of the dominant transactions to detect cycles. The cycles determine circular waitings of transactions. Algorithm 3 deals with cycles between transactions.

The local dependency graphs are used by the Participants Managers to detect the cycles. A `WaitingCycleCheck` token is sent to the coordinators of the dominant transactions. If a coordinator of the dominant transaction does not have any waiting web

```
Algorithm 2 complete()
execute service oi for transaction ti;
switch service results of r do
    case r is cancel
         roll back;
         deleteFromSerializationGraph(ti, oi);
         send message CANCEL
    case r is exit
         send message EXIT;
    case r is completed
        prepare commit;
        check dependencies in SG
        if (ti has at least one dominant transaction td)
        do
            send message WAIT;
            CycleCheck(); // see algorithm 3
        od
        else send message COMPLETED
end switch
wait for the next message from coordinator;
Exception When Fault
   send message FAULT;
    wait for the next message from coordinator;
```

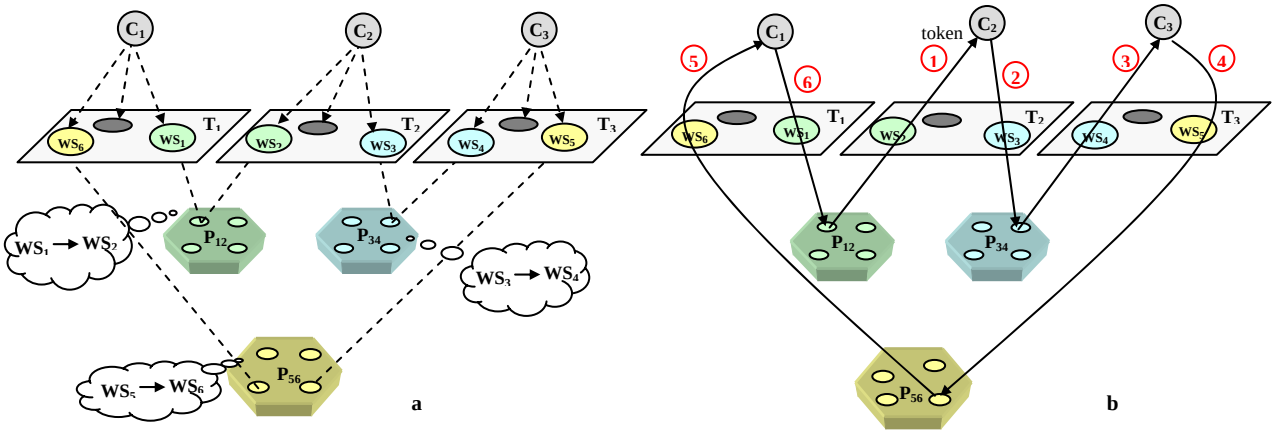

Figure 7. a) Three Transactions Running into a Waiting Cycle
b) Using a WaitingCycleToken to Detect the Cycle

Algorithm 3 *CheckCycle()*

```
TD = {};  //dominant transaction
TD ← getDominantTransactions(ti; oi);
for all ti ∈ TD do
    send CycleCheck(ti, oi) token to ci ∈
Coord_of(ti);
    wait for message from ci or UNTIL timeout;
end for
When Message from a coordinator received
  switch message type of m do
      case m is CycleCheck(ti, oi)
          if (own token received) do
              prepare commit;
              send message COMPLETED to ci of ti;
          od
          else forward token to all ti ∈ TD
      case m is NoCycle(ti, oi)
          wait;
      case m is timeout
          compensate();
          send message COMPENSATED;
  end switch
```

services, he sends back a `NoWaitingCycle` token. Otherwise, it propagates the `WaitingCycleCheck` token to all of its waiting web services and so on.

A cycle is detected when the issuing provider has received back his own `WaitingCycleCheck` token from the coordinator, who originally started the completion phase (Figure 7.b). Once a waiting cycle was detected, the Participants Manager can safely resolve this cycle by confirming the readiness to complete so that the coordinator, who started the completion procedure, can close its participant and as a result of this, its dependent coordinator will be able to lose as well. Applying this to our example scenario in Figure 7.b, will lead to having (the participant of) WS_1 sending a completed message to its coordinator C_1 (which initially started the completion procedure which triggered the waiting cycle check). C_1 then will commit T_1 and (the participant of) WS_6 will inform its Participants Manager P_{56}, which in turn will remove it from its local dependency graph and inform all its dependents (WS_5). WS_5 will then inform C_3 about its readiness to close, which will enable the completion of C_3. By closing T_3, WS_3 will be informed via P_{23}. At this point C_2 will have no more dominant transactions, so it can commit safely as well.

7. Implementation

We implemented part of the WS-Coordination and the WS-Transaction protocol, focusing on the `BusinessActivityWithCoordinatorCompletion`, and extended it with our protocol as basis for our prototype and evaluation. On the web service provider side (server-side), we implemented the Participants Manager component and extended the participant's functions to be able to communicate with it. On the client-side, we extended the coordinator's functions to be able to handle the new participant state, i.e. the `waiting` state and to respond to a CheckWaitingCycle message. In addition to the standard message from the original WS-Coordination/WS-Transaction protocols, the coordinator can now receive one extra message from the participant: a `wait` message. Different from the current standard (a `compensated` message can only be received as a response to a `compensate` command), a dependent coordinator can now receive a `compensated` message from a waiting participant, if the dominant coordinator rolled back. It also can receive a CheckWaitingCycle message from a Participants Manager and respond either by forwarding the message to all its waiting participants or by sending a NoWaitingCycle. The coordinator should also be able to forward a NoWaitingCycle message back to the Participants Manager which originally sent the CheckWaitingCycle message; therefore a coordinator must be able to map the received tokens to the original senders. Figure 8 shows the WSDL description of both the Coordinator and the Participants Manager interfaces.

For prototyping we used 3 machines representing three different web service providers. Every machine was equipped with Apache Tomcat 5.5 as an application server and Apache Axis 1.2 as a SOAP engine. We implemented the provider part in our proposed protocol on each machine. Each server provides 2 web services, which simulate the functions of crediting and debiting a user account. A user account is represented by a text file which holds the current status of the account. Each web service was provided with a compensation operation that can be invoked to undo the job done by the primary operation.

For our experiment we ran 3 transactions on 3 client machines. Each transaction transfers money from one account to another one by invoking 2 web services from 2 different providers. We distributed the web services among the transactions such that there is a dependency relation between every 2 transactions. Table 1 summarizes our experiment by comparing the final outcome of two runs: without and with

concurrency control mechanism. In the latter case, the dependent transactions of a failed dominant transaction were informed and were able to compensate. In the case of a failure-free execution the 3 transactions ran into a waiting cycle and the Participants Managers were able to detect, report and resolve the waiting cycle without the need to any direct communication between different transactions, nor between different providers.

Table 1: Experiment Results

a) Without Concurrency Control

Time	WS	Peter	Mohammad	Wolfgang	Result
t0		0	0	0	
t1	*T1.credit(P, 300)*	300			succeeds
t2	*T2.debit(P,200)*	100			succeeds
t3	*T2.credit(M,200)*		200		succeeds
t4	*T3.debit(M,100)*		100		succeeds
t5	*T3.credit(W,100)*			100	succeeds
t6	*T1.debit(W,300)*				fails
t7	*T1.compensate(t1)*	-200			
tfinal		-200	100	100	

b) With Concurrency Control

Time	WS	Peter	Mohammad	Wolfgang	Result
t0		0	0	0	
t1	*T1.credit(P, 300)*	300			succeeds
t2	*T2.debit(P,200)*	100			succeeds
t3	*T2.credit(M,200)*		200		succeeds
t4	*T3.debit(M,100)*		100		succeeds
t5	*T3.credit(W,100)*			100	succeeds
t6	*T1.debit(W,300)*				fails
t7	*T1.compensate(t1)*	-200			
t8	*T2.compensate(t2)*	0			
t9	*T2.compensate(t3)*		-100		
t10	*T3.compensate(t4)*		0		
t11	*T3.compensate(t5)*			0	
tfinal		0	0	0	

8. Discussion

Our approach for supporting concurrency control in web services environment can be built on top of well established standards, namely the WS-Coordination and WS-Transaction Protocols. In our approach we delegate the concurrency control management to the service provider instead of adding more responsibilities and duties to the coordinator of the transaction on the client side. We believe that the coordination of a set of web services in a transactional process must be conducted by the client who is running this process and benefiting from it. While, on the other hand, the management of concurrent access to local resources of a service provider by different independent transactions is a task that has to be done by the server itself, since it has the required knowledge about the possible conflicts and can keep track of all web service invocation requests from the remote transactions coordinators. The use of the concept of a Participants Manager enabled us to achieve a globally correct execution without the need to direct communications between independent transactions, which (more likely) are executed by different parties and may involve some mission-critical information.

However, this approach has its cost. Compared to other approaches that rely on such direct communications between the transactions coordinators, our approach requires 2 times the number of exchanged messages to reach globally correct execution. The reason is that we replace each single direct message between two coordinators C_1 and C_2 by two messages: one message from C_1 to the common provider P_{12}, and another message from P_{12} to C_1. It is a trade-off relation between the cost in terms of the number of exchanged messages and the security and privacy properties that can be ensured using our approach.

9. Conclusion and Further Work

We have described an extension to the WS-Transaction Protocol for concurrency control in transactional web services environments. The protocol ensures consistency of data when independent business transactions access the data concurrently under the relaxed transaction properties. The protocol is based on transaction dependency graphs maintained at the service provider side. We have shown several algorithms implementing the protocol. We have implemented such a protocol on top the current web service transactions standard [1, 2]. Such an extension on the web service provider side is simple to achieve as it can be implemented as an integral part of the service invocation mechanism. It has also an advantage from an engineering point of view as it does not change the way consumers or clients of web services have to be programmed. In addition, it avoids direct communication between transaction coordinators and thus preserves security by keeping the information about business transactions restricted to the coordinator responsible for these transactions.

In future work we plan to run further experiments with the proposed protocol and evaluate how it

performs with regard to throughput and reliability in case of many commits. We also plan to investigate different commit strategies in such an environment, as well as error recovery and compensation strategies.

10. References

[1] Arjuna Technologies, BEA Systems, Hitachi Ltd., IBM, IONA Technologies, and Microsoft, *Web services atomic transaction*, 2005, published at ftp://www6.software.ibm.com/software/developer/-library/WS-AtomicTransaction.pdf

[2] Arjuna Technologies, BEA Systems, Hitachi Ltd., IBM, IONA Technologies, and Microsoft, *Web services business activity framework transaction* , 2005, published at ftp://www6.software.ibm.com/software/ developer/library/WS-BusinessActivity.pdf.

[3] Arjuna Technologies, BEA Systems, Hitachi Ltd., IBM, IONA Technologies, and Microsoft Corporation. *Web services coordination* 2005, published at ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[4] OASIS *Business transaction protocol*, 2004, published at http://www.oasis-open.org/committees/documents.php?wg_abbrev=business-transaction.

[5] Arjuna Technologies, BEA Systems, Hitachi Ltd., IBM, IONA Technologies, and Microsoft.
Web services composite application framework (ws-caf), 2003, published at http://developers.sun.com/techtopics/webservices/wscaf.

[6] S. Choi, H. Jang, H. Kim, J. Kim, S. Kim, J. Song, and Y. Lee. *Maintaining consistency under isolation relaxation of web services transactions.*
In Proc. of WISE 2005, New York, USA, Nov. 2005.

[7] K. Haller, H. Schuldt, and C. Türker. *Decentralized coordination of transactional processes in peer to peer environments.* ACM Press, in Proc. of the 14th ACM Intl. Conference on Information and Knowledge Management (CIKM 2005), pages 36--43, Bremen, Germany, Nov. 2005.

[8] W3C *Simple object access protocol* (soap) 1.2, 2003. W3C Note, http://www.w3.org/TR/soap12-part1/.

[9] W3C *Web service description language* (wsdl) 1.1, 2001. W3C Note, http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[10] F. Curbera, Y. Goland, J. Klein, F. Leyman, D. Roller, S. Thatte, and S. Weerawarana. *Business process execution language for web services* (bpel4ws) 1.0, August 2002. W3C Note, http://www.ibm.com/developerworks/library/ws-bpel.

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

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

[13] Özsu, M.T., Valduriez, P., *Principles of Distributed Database Systems*, 2nd Edition, Prentice Hall (1999)