

# Transaction Concurrency Control in Web Service Environment

Mohammad Alrifai  
L3S Research Center,  
University of Hannover,  
Expo Plaza 1, 30539 Hannover, Germany  
alrifai@l3s.de

Peter Dolog  
L3S Research Center,  
University of Hannover,  
Expo Plaza 1, 30539 Hannover, Germany  
dolog@l3s.de

Wolfgang Nejdl  
L3S Research Center,  
University of Hannover,  
Expo Plaza 1, 30539 Hannover, 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.*

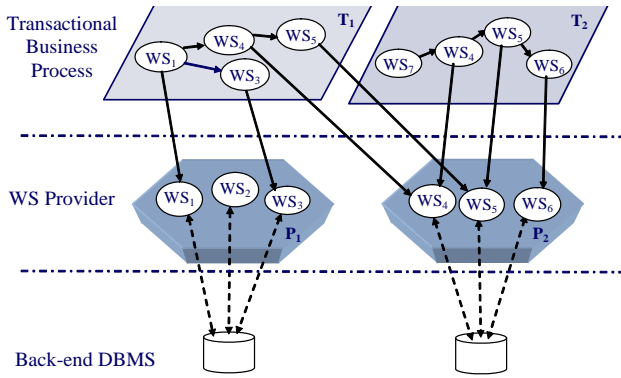
## 1. Introduction

Web services are distributed independent computing units that provide a Business-to-Business interface based on standards like SOAP [10] and WSDL [9]. They allow the integration and collaboration of different business applica-

tions running on different heterogeneous back-end systems. BPEL4WS [6] 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. The traditional ACID (Atomicity, Consistency, Isolation and Durability) properties of a transaction management system have to be relaxed for web services-based transactions, since they are based on resources locking mechanisms which are impossible or 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. 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.

Because of the Atomicity-relaxation property, 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 ser-



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

vice within the same transaction. Therefore, support for recovery 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 to the problem of transaction management in web service environment either address one part of the problem, the recovery problem [2, 3, 4, 8, 1] or they involve several independent transaction coordinators communicating with each other [5, 7].

While these approaches are important first steps, they neglect that using these protocols possibly mission-critical information is disclosed to parties which should not have access to this information. 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 (peer) avoiding direct communication between third party transactions;
- Peers know the transactions which invoked their operations so they can contact them immediately when a dependency check should be performed;
- As dependent transactions will be informed (by the common participant) as soon as the relevant web ser-

vice 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 [2, 3, 4], BTP [8] and WS-CAF [1]) 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. 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 [5, 7]. In [5] 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 type of airplane offered for the trip for which the tickets are offered to. 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 thus provides the tickets to the customer as available. 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. Also they are long running processes so locking for the whole transaction running time would not be 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) 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. Current web service transactional management protocols standards yet do not provide such possibility.

### 4 WS-Coordination and WS-Transaction Protocols

Unlike the OASIS Business Transaction Protocol (BTP) which is aimed to representing and seamlessly managing complex business-to-business transactions over the Internet, both WS-Coordination and WS-Transaction 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

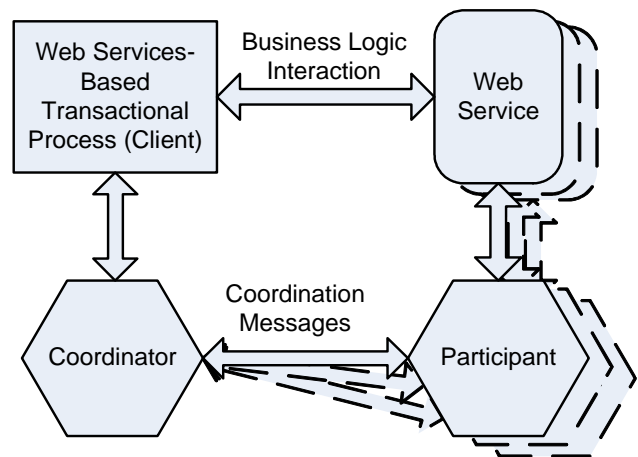


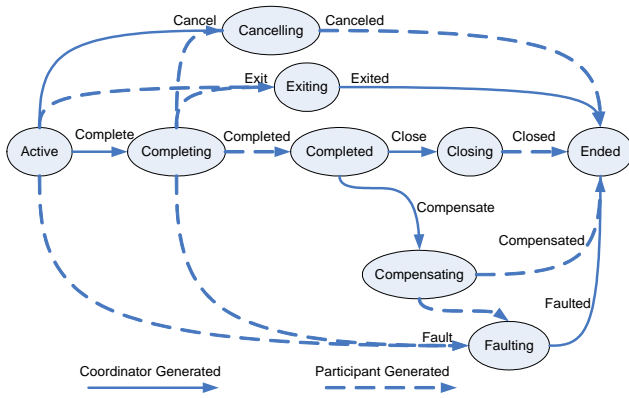
Figure 2. WS-Coordination Architecture

WSDL, WS-Addressing, Web Services Security, and WS-Policy.

**WS-Coordination.** The WS-Coordination specification 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 (Fig. 2).

**WS-Transaction.** The WS-Transaction specification 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 [WS-Tx].

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 to either commit or cancel. 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.



**Figure 3. BusinessAgreementWithCoordinatorCompletion abstract state diagram**

**WS-BusinessActivity.** This specification defines two specific agreement coordination protocols for the Business-Activity 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 providers point of view. The protocol operates on transaction schedules. Transaction *schedule* is a set of transactions in partial order where some of the transactions may be executed concurrently.

Two operations are in *conflict* in a transaction schedule if they access the same resource and at least one of them is influencing a result of the both operations (for example by writing a data item used to compute the operations output).

A *serialization graph* is a graph consisting of transactions as nodes and edges representing relations between transactions in conflict.

We use the notion of the serialization graph for describing a *completion dependency*; i.e. a dependency which prevents an operation in a transaction to complete when it reads data updated by other operations in other transactions and those operations may still roll back. The transactions which consist of such dependent operations are called *dependent transactions*. The transactions, which consist of operations which other operations depend on are called *dominant transactions* with regard to the transactions with dependent transactions.

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 dependant 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 dependency 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 on should be able to detect waiting cycles and resolve them.

The key issues are:

- If a dependency is detected, how can we detect circular waiting?
- When circular waiting is detected, how can it be resolved?

## 6 A Support for Concurrency Control for Web Services

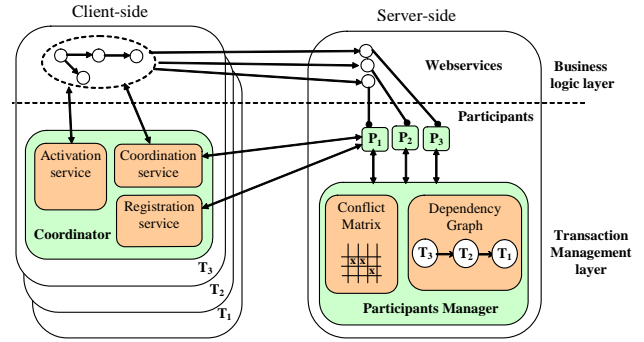
**Transaction Dependencies Management.** The transaction aware web service environment consists of set of ser-

vice providers (peers) and a set of transaction coordinators. Each service provider provides one or more operations as web service. Client may initiate transactions with operations residing on several web services. The transactions are running as transactional processes coordinated by the transaction coordinators. When a transaction is initiated it is assigned to a coordinator.

Due to relaxed atomicity and isolation properties, operations from different transactions can run into conflicts. We use the notion of transaction dependencies introduced in the previous section to maintain information about the conflicts. The conflict dependency matrix is maintained at each provider for each transaction which access the operations provided by the provider. Besides transaction dependencies, each web service maintain information about its state with respect to the completion.

**Participant 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 a conflict matrix, which was 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  $N \times N$  matrix, where  $N$  is the total number of the provide web service operations that can be accessed via the web. 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. The nodes of this graph represent the running transactions and the directed edge between two nodes means that there is a dependency relation between these two nodes such that the edge points from the dependant transaction to the dominant one. 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 mes-



**Figure 4. Components of a transaction aware web service environment**

sage) 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 dependant 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).

**The Protocol.** A dependency is detected when another transaction invokes an operation which is in conflict with operations invoked by already running transactions. The dependency is stored at the peer. Algorithm 1 describes the main execution thread at the provider side.

A coordinator of the transaction which invoked the operation is informed that the operation is *waiting*. The operation waits until it receives a *close* or *compensate* message from the dominant transactions. The close message is propagated to the transaction where the operation belongs to. Algorithm 2 describes how the completing phase is performed at the provider side.

If a compensate message is received, the operation compensates web services which invoked the operation. Peer also informs other transactions that it will compensate the operation they invoked. Additional messages for finishing, closing and detecting faults are introduced respectively.

Figure 5 depicts an example of such a transaction aware web service environment. It depicts three transaction coordinators coordinating three transactional processes. 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  $WS_6$ .  $P_{12}$  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

---

**Algorithm 1** Service Provider

---

```
SG = {} {local serialization graph}
O = {o1, . . . , oi} {operations at provider} {body of the
service provider execution}
activate(oi);
addToSerializationGraph(ti, oi);
wait for next message from coordinator;
When Message from a coordinator received
switch message type of m do
  case m is cancel
    roll back;
    send message CANCELED;
  case m is complete
    complete();//see next algorithm
  case m is close
    commit;
    deleteFromSerializationGraph(ti, oi);
    send message CLOSED;
  case m is compensate
    compensatate();
    send message COMPENSATED;
  case m is exited
    roll back;
    deleteFromSerializationGraph(ti, oi);
    send message CLOSED;
  case m is faulted
    roll back;
    deleteFromSerializationGraph(ti, oi);
    send message CLOSED;
end switch
Exception When Fault
  roll back;
  send message FAULTED;
  wait for next message from coordinator;
```

---

( $T_1$ ) 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.

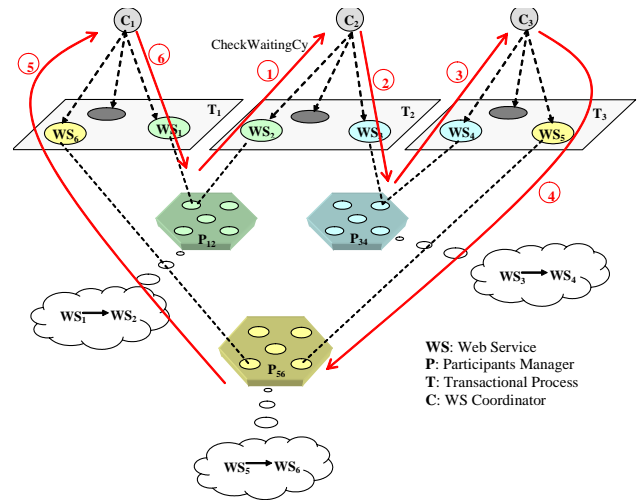
---

**Algorithm 2** complete()

---

```
execute service oi;
switch service results of r do
  case r is cancel
    roll back;
    send message CANCEL
  case r is exit
    prepare roll back;
    send message EXIT;
    wait for the next message from coordinator;
  case r is fault
    prepare roll back;
    send message FAULT;
    wait for the next message from coordinator;
  case r is completed
    prepare commit;
    send message COMPLETED
    wait for next message from coordinator;
  case r is waiting
    send message WAIT;
    CycleCheck();
end switch
```

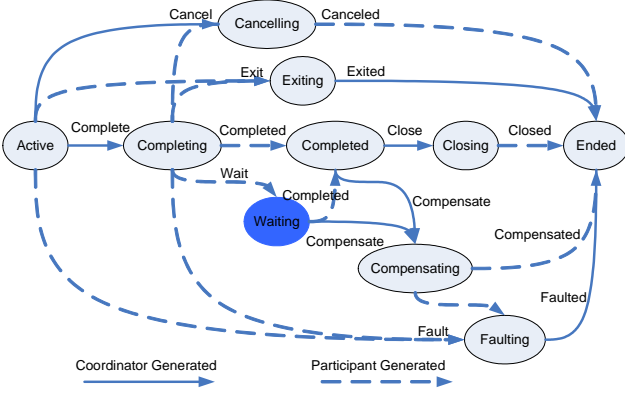
---



**Figure 5. Three transactions running into a waiting cycle**

**Cycle Detection.** As shown on Fig. 5 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  $T_3$  is a dominant



**Figure 6. Participant abstract state diagram including the new waiting state**

of  $T_2$  and  $T_1$  is a dominant transaction of  $T_3$ . If  $T_1$  starts its completion phase at some point of the time, its Coordinator  $C_1$  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  $P_{12}$  and will send a wait message to  $C_1$ . The same holds for  $T_2$  and  $T_3$ , hence it will turn out that all coordinators will run into a waiting cycle since  $C_2$  is waiting for  $C_3$  and  $C_3$  is waiting for  $C_1$ , which is waiting for  $C_2$ . 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 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 services, he sends

---

### Algorithm 3 CycleCheck()

---

```

TD = {}; // dominant transaction
TD ← getDominantTransactions( $t_i, o_i$ );
for all  $t_i \in TD$  do
    send CycleCheck( $t_i, o_i$ ) token to  $c_i \in \text{Coord\_of}(t_i)$ ;
    wait for message from  $c_i$  or UNTIL timeout;
end for
When Message from a coordinator received
switch message type of  $m$  do
    case  $m$  is CycleCheck( $t_i, o_i$ )
        prepare comit;
        send message COMPLETED to  $c_i$  of  $t_i$ ;
        wait for next message from coordinator;
    case  $m$  is NoCycle( $t_i, o_i$ )
        wait for next message from coordinator;
    case  $m$  is timeout
        compensate();
        send message COMPENSATED;
        wait for next message from coordinator;
    end switch

```

---

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. 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 Fig. 5, 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, extended with our protocol as basis for our prototype and evaluation. Both client-side (coordinator) services and server-side (participant) ser-

vices were implemented. We further implemented several transaction-aware web services and made them available on one machine, which represents the web service provider. Each web service has one transactional operation, which reads data from a text file, processes it and writes back results into the file. Each web service was provided with a compensation operation that can be invoked to undo the job done by the primary operation. The participant's part of the `BusinessActivityWithCoordinatorCompletion` transaction protocol was deployed on the provider machine such that whenever a web service is invoked, an instance of the participant is initiated and assigned to that web service.

We ran several transactional processes on some other machines (clients), such that each transactional process invokes several web services from the web service provider machine. The Coordinator's part of the `BusinessActivityWithCoordinatorCompletion` transaction protocol was implemented on every client machine. The execution of a client program, which involves a transactional process, starts by initiating a transaction coordinator. The coordinator in turn creates a transaction context and returns it back to the client program, which appends it to the parameter list passed to the web services when they are invoked. The corresponding participant of the invoked web service uses this transaction context to communicate with the coordinator according to the transaction protocol.

We extended the original protocol by adding the proposed functions and components to support the concurrency control for web services. On the web service provider side (server-side), we implemented the Participants Manager component and extended the participants functions to be able to communicate with it. On the client-side, we extended the coordinators functions to be able to handle the new participant state, 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 two extra messages from the participant: a wait message and a compensated message. It also can receive a `CheckWaitingCycle` message and respond either by forwarding the message to all its waiting participants or by sending a `NoWaitingCycle`.

Figure 7 shows the WSDL description of both the Coordinator and the ParticipantsManager interfaces. For prototyping we ran 3 transactions on 3 client machines. Each transaction invokes several web services. We distributed the web services among the transactions such that there is a dependency relation between every 2 transactions (to form a waiting cycle).

For the peers hosting the web services, we used 3 machines representing three different organizations. Every machine was equipped with Apache Tomcat 5.5 as an application server and Apache Axis 1.2 as a SOAP engine. We

```

<!-- Coordinator Interface -->
<wsdl:portType
  name="CoordinatorPortType">
  <wsdl:operation name="Wait">
  <wsdl:input message="wstx:Wait"/>
  </wsdl:operation>
  <wsdl:operation
    name="Compensate">
  <wsdl:input
    message="wstx:Compensate"/>
  </wsdl:operation>
  <wsdl:operation
    name="CheckWaitingCycle">
  <wsdl:input
    message="wstx:CheckWaitingCycle"/>
  </wsdl:operation>
  <!--other operations from
    the standard go here -->
  </wsdl:portType>

<!-- Participant Interface -->
<wsdl:portType
  name="ParticipantPortType">
  <wsdl:operation
    name="CheckWaitingCycle">
  <wsdl:input
    message="wstx:CheckWaitingCycle "/>
  </wsdl:operation>
  <wsdl:operation
    name="NoWaitingCycle">
  <wsdl:input
    message="wstx: NoWaitingCycle "/>
  </wsdl:operation>
  <!--other operations
    from the standard go here -->
  </wsdl:portType>

```

**Figure 7. WSDL descriptions of Coordinator and ParticipantsManager**

implemented the provider part in our proposed protocol on each peer and ran it on each peer.

To simulate the failure situation of a given web service, after a randomly set number of calls, the peer holding the web services should report the called web service as failed.

The three transactions completed successfully and was able to solve the dependency problem of concurrent transactions as well as to detect and resolve the waiting cycle without the need to any direct communication between different transactions, nor between different peers (organizations).



## 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 is 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 [2, 3]. 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.

## References

- [1] 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>.
- [2] Arjuna Technologies, BEA Systems, Hitachi Ltd., IBM, IONA Technologies, and Microsoft. Web services atomic transaction (ws-atomictransaction), 2005. published at <ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf>.
- [3] Arjuna Technologies, BEA Systems, Hitachi Ltd., IBM, IONA Technologies, and Microsoft. Web services business activity framework transaction (ws-businessactivity), 2005. published at <ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf>.
- [4] Arjuna Technologies, BEA Systems, Hitachi Ltd., IBM, IONA Technologies, and Microsoft Corporation. Web services coordination (ws-coordination), 2005. published at <ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf>.
- [5] 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.
- [6] 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>.
- [7] K. Haller, H. Schuldt, and C. Trker. Decentralized coordination of transactional processes in peer to peer environments. In *Proc. of the 14th ACM Intl. Conference on Information and Knowledge Management (CIKM 2005)*, pages 36–43, Bremen, Germany, Nov. 2005. ACM Press.
- [8] OASIS. Business transaction protocol, 2004. published at [http://www.oasis-open.org/committees/documents.php?wg\\_abbrev=business-transaction](http://www.oasis-open.org/committees/documents.php?wg_abbrev=business-transaction).
- [9] W3C. Web service description language (wsdl) 1.1, 2001. W3C Note, <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.
- [10] W3C. Simple object access protocol (soap) 1.2, 2003. W3C Note, <http://www.w3.org/TR/soap12-part1/>.
- [11] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. Morgan Kaufmann, 2001.