Available online at www.sciencedirect.com

ScienceDirect

Information and Software Technology 50 (2008) 684–696

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

A framework for ensuring consistency of Web Services Transactions

Seunglak Choi [a,*], Hangkyu Kim [a], Hyukjae Jang [a], Jungsook Kim [b], Su Myeon Kim [c], Junehwa Song [a], Yoon-Joon Lee [a]

[a] *Korea Advanced Institute of Science and Technology, 371-1 Guseong-Dong, Yuseong-Gu, Daejeon, Republic of Korea*
[b] *Electronics and Telecommunications Research Institute, 161 Gajeong-dong, Yuseong-gu, Daejeon, Republic of Korea*
[c] *Samsung Advanced Institute of Technology, Mt. 14-1, Nongseo-dong, Giheung-gu, Younggin-si Gyunggi-do, Republic of Korea*

Received 3 July 2006; received in revised form 1 July 2007; accepted 17 July 2007
Available online 6 August 2007

Abstract

For efficiently managing Web Services (WS) transactions which are executed across multiple loosely-coupled autonomous organizations, isolation is commonly relaxed. A Web service operation of a transaction releases locks on its resources once its jobs are completed without waiting for the completions of other operations. However, those early unlocked resources can be seen by other transactions, which can spoil data integrity and cause incorrect outcomes. Existing WS transaction standards do not consider this problem. In this paper, we propose a mechanism to ensure the consistent executions of isolation-relaxing WS transactions. The mechanism effectively detects inconsistent states of transactions with a notion of an *end-state dependency* and recovers them to consistent states. We also propose a new Web services Transaction Dependency management Protocol (WTDP). WTDP helps organizations manage the WS transactions easily without data inconsistency. WTDP is designed to be compliant with a representative WS transaction standard, the Web Services Transactions specifications, for easy integration into existing WS transaction systems. We prototyped a WTDP-based WS transaction management system to validate our protocol.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Web services; Transaction model; Transaction management protocol; Consistency

1. Introduction

Major IT organizations such as Amazon, Google, and e-Bay have been migrating their interfaces for business partners to service-oriented architectures using the Web Services (WS) technology. Due to its flexibility and dynamicity, WS allows organizations to easily integrate services across different organizations as well as within organizations [1,4,13,15,18]. Such WS-based integrated applications should guarantee consistent data manipulation and outcome of business processes running across multiple loosely-coupled organizations. Thus, WS technologies should be extended to equip with transaction-processing functionalities.

There are three proposals for protocols to extend the WS with transaction-processing capabilities, i.e. Web Services Transactions specifications [10], Business Transaction Protocol (BTP) [16], and WS-CAF [2]. Also, commercial transaction management systems have been developed implementing these protocols. For efficient processing, these WS-based transaction protocols relax the *isolation property*[1] rather than use strictly exclusive locking mechanisms such as the two-phase commit and the strict two-phase locking[2] [9]. In contrast to traditional transactions, WS transactions live long – e.g., a few hours or days. Thus,

* Corresponding author.
E-mail addresses: slchoi@dbserver.kaist.ac.kr, seunglak@gmail.com (S. Choi), hkkim@dbserver.kaist.ac.kr (H. Kim), hjjang@nclab.kaist.ac.kr (H. Jang), jungsook96@etri.re.kr (J. Kim), sumyunkim@samsung.com, smkim@nclab.kaist.ac.kr (S.M. Kim), junesong@nclab.kaist.ac.kr (J. Song), yjlee@dbserver.kaist.ac.kr (Y.-J. Lee).

[1] Isolation is one of the well-known transaction properties called ACID – Atomicity, Consistency, Isolation, and Duration.

[2] Those schemes maintain consistency through holding locks on all resources until the completion of a transaction. Thus, a resource cannot be released before all other resources are ready to be released.

if the strict locking mechanisms are used, an organization may not be able to access its resources even for a few days, until other organizations complete their works and release their resources. Relaxing the isolation property, an organization participating a transaction completes its work without concerning the status of other organizations' works. The organization releases its resources and continues its jobs for other transactions. In fact, most models for long-lived transactions proposed in literature [2,6,8,19,21,16] relaxed the isolation property.

In this paper, we argue that the isolation relaxation introduces a serious inconsistency problem. Consider a situation where a participant fails its transaction after releasing a shared resource and assume that other participants have already read the resource and proceeding their own transactions (see Section 2 for details). Such a situation implies that different participants hold different states of the same resource, resulting in a possibly serious problem. Considering that WS transactions will get more and more prevalent, such a situation may occur quite frequently, being a major blocking factor for the WS transaction usage. However, existing WS transaction models and managing systems as well as the long-lived transaction models do not address this inconsistency problem. After all, organizations have to solve the problems by themselves. However, the lack of pre-defined protocols among organizations makes it hardly possible since WS transactions are dynamically created across any Web services hosted by many loosely-coupled organizations.

We propose a mechanism to ensure the consistent executions of the isolation-relaxing transactions. The mechanism effectively identifies the transactions in an inconsistent state with a notion of an *end-state dependency*. The notion is a relationship between two transactions in which one transaction's failure incurs the inconsistent state of the other transaction. Once the inconsistent transactions are identified, the mechanism recovers those transactions to the previous consistent states and optionally further re-executes them. In the mechanism, a transaction should delay its end until its related transactions are ended, i.e. they are guaranteed never to abort. Occasionally, the delayed completion incurs *circular waiting*, in which two or more transactions is waiting for the other to be ended and thereby no one can be ended. The mechanism provides a graph-based method to detect circular waiting and defines a condition to safely end the involved transactions without inconsistency.

We also propose a new *Web services Transaction Dependency management Protocol* (WTDP). WTDP makes it possible for participants to identify and resolve occurrences of end-state dependency. Upon its occurrence, using the protocol, participants automatically deliver related information to other participating organizations. Any changes affecting the status of the dependency are continuously monitored and processed, thus each organization is notified of the status of related transactions. In addition, WTDP detects and resolves circular waiting by a token-based approach, which is executed on dependency information distributed over multiple sites. Adopting the protocol, organizations do not need to concern the management of the dependencies. They only need to process transactions as they did according to the status notification of relevant transactions. In addition, being compliant with the de-facto WS transaction standards, WTDP is easily implemented and deployed within existing WS transaction management systems.

The rest of this paper is organized as follows: Section 2 shows a motivating scenario of dependencies among WS transactions. In Section 3, we review related work and the WS transactions specifications. Then, we propose a mechanism to guarantee consistent executions of transactions in Section 4, and WTDP in Section 5. Section 6 presents a prototype system supporting WTDP. Finally, we conclude this work in Section 7.

2. Motivating scenario

Suppose there are five companies: furniture maker, wood distributor, steel distributor, lumber mill, and shipping company. The furniture maker manufactures furniture using both wood and steel which are supplied by the wood and steel distributors, respectively. The wood distributor buys wood from the lumber mill. Wood is delivered to the wood distributor by the shipping company. All the companies implement and use Web services for the above processes.

The wood distributor and the lumber mill agree on a vendor managed inventory (VMI) contract, which is an advanced inventory management technique [20]. Under the VMI contract, producers are responsible for managing and replenishing consumers' inventories. The producer periodically receives electronic data that tell him the stock levels. By analyzing those data with statistical methodologies, the producer forecasts and maintains the correct inventory. Therefore, under VMI, the producer is in total control of the timing and size of the order to replenish the inventory, not the distributor. The key benefits of VMI are that (1) consumers can reduce the overhead of inventory management and (2) producers replenish the inventory level in a timely manner. In our scenario, the lumber mill and the wood distributor correspond to VMI producer and VMI consumer, respectively. The lumber mill therefore continuously checks the inventory of the wood distributor and delivers wood whenever the amount of wood in the inventory goes below an agreed level of stocks.

Fig. 1 shows a snapshot of the internal process of three companies when the furniture maker receives a purchase order from a customer who needs 50 units of woods and steels. The furniture maker issues ordering transaction (ORDER_T) to acquire 50 units of wood and 50 units of steel. The wood order is processed and committed instantly, but the processing of steel order is delayed due to an internal problem of the steel distributor. At that moment, the lumber mill issues VMI transaction (VMI_T). It checks the inventory of the wood distributor and realizes

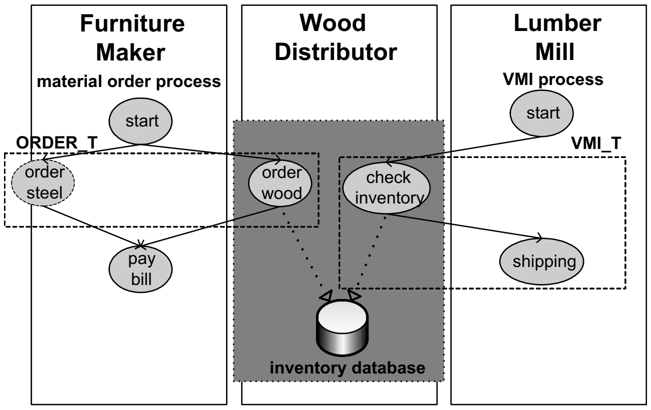


Fig. 1. The occurrence of dependency between two Web Services transactions.

that the total stock is less than the agreed amount of units (here 100 units). VMI_T automatically initiates supplying 50 units of wood. Let us assume that the order of 50 units of steel is aborted. Then, the supply of wood from the wood distributor to the furniture maker will be aborted. In that case, the wood distributor does not need the extra 50 units of wood any more. Thus, the supply of wood from the lumber mill to the wood distributor should be aborted instantly if possible.

In the level of transactions, VMI_T is said to be dependent upon ORDER_T; if ORDER_T fails, VMI_T should be aborted. By showing the intermediate value of the total stock (i.e. 50) to VMI_T, which is inherent due to the relaxation of isolation, the status of ORDER_T determines the status of VMI_T. ORDER_T and VMI_T are considered as completely independent transactions by current Web services transaction standards. There are no model to express the dependency and no protocols to handle such dependencies.

3. Related work

3.1. Business transaction models

Several isolation-relaxing transaction models have been introduced since late 1980s. Garcia-Molina and Salem [8] addressed the performance problem of long-lived transactions. To alleviate this problem, they deal with a long-lived transaction (called *saga*) as a sequence of sub-transactions and permit each sub-transaction to be committed, irrespective of the commitment of other sub-transactions. They also addressed the concept of *compensation* to amend a partial execution. Weikum and Schek [19] proposed an open transaction model, in which a large transaction is decomposed into nested sub-transactions. These sub-transactions can be also committed unilaterally. A flexible transaction model [7,11,21] was proposed in the context of multi-database systems. It allows sub-transactions to be committed without waiting the completion of delayed sub-transactions. Note that all the transaction models above do not consider the inconsistency problem caused by concurrently executing isolation-relaxing transactions.

Nowadays, isolation-relaxing transaction models have been realized for business transactions under the Web services environment. The WS transactions specifications [10] are a de-facto standard of a framework and protocols to implement isolation-relaxing transactions on Web services, proposed by IBM, MS, and BEA. The Business Transaction Protocol (BTP) [5,16] was proposed by the OASIS business transactions technical committee, and also relaxes isolation of transactions. WS-CAF [2], proposed by Arjuna Technologies Limited et al., consists of three protocol layers which support long-running, complex business process transactions in which isolation relaxation is a basic concept. Note that, however, like the prior transaction models, these protocols do not handle the inconsistency problem of isolation-relaxing transactions.

Some research, although not focusing on the inconsistency problem, has recently been done in the area of Web Services Transactions. Schmit and Dustdar [17] briefly discussed the inconsistency problem of isolation-relaxing transactions, however, did not suggest any solutions for that problem. Mikalsen et al. [12] introduced a middleware for building and executing transactional compositions from Web services with implicit transactional semantics. Papazoglou [14] reviews current research and standard activities of a Web services transaction framework, which is slightly outdated due to the on-going research activities.

3.2. WS transactions specifications

The WS transactions specifications describe the framework for coordinating transactions running across multiple Web services. Since WTDP is designed as an extension of the WS transactions specifications, we briefly review the specification to facilitate the understanding of WTDP.

The specifications include the WS-Coordination specification, which defines an extensible framework (commonly called a coordinator) for providing protocols that coordinate the activities of distributed transactions. The specifications also provide two coordination types, WS-AtomicTransaction and WS-BusinessActivity, for simple short-lived transactions and complex long-lived business activities, respectively. By using those two specifications, the coordinator manages WS transactions. The coordinator consists of the following services, which is illustrated in Fig. 2. All these services are provided as Web services.

- Activation service (specified in WS-Coordination) – operations that enable an application to create a coordination instance or context.
- Registration service (specified in WS-Coordination) – operations that enable an application and participants to register for coordination protocols.
- Protocol service (specified in WS-AtomicTransaction/BusinessActivity) – a set of coordination protocols.

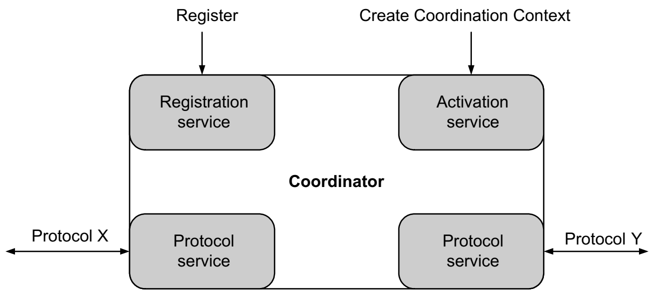


Fig. 2. Coordinator architecture.

When an organization wants to create a new WS transaction, it requests and receives a coordination context from its coordinator using the Activation service. The organization notifies all the participants of the WS transaction by forwarding the coordination context to them. Then, all the participants and the organization (WS transaction initiator) register their own transaction protocols, which are specified in the WS-AtomicTransaction/BusinessActivity specifications, to the coordinator using the Registration service. Then, the coordinator manages the transaction by exchanging messages with the transaction initiator and the participants via the Protocol service.

4. Maintaining consistency

In this section, we present a mechanism to ensure the consistent executions of isolation-relaxing Web Services Transactions. In Section 4.1, we first briefly introduce an underlying transaction model on top of which our mechanism is built. We then illustrate the inconsistency problem concretely in Section 4.2. In Section 4.3, we then formally define the end-state dependency, which is used to determine transactions affected by an abort of a given transaction. In Section 4.4, we describe how to handle the dependency. Finally, we discuss circular waiting which occurs when two or more transactions are waiting for other transactions to be ended, and also describe a scheme to handle circular waiting in Section 4.5.

4.1. Underlying transaction model

Our mechanism is designed for the transactions based on the business activity model of the WS-Transaction specifications. In that model, a transaction consists of multiple operations on a collection of Web services. The model allows any number of hierarchical nesting levels, i.e. an operation is nested within a transaction, which in turn can be nested within another transaction and so on (see an example of a transaction hierarchy shown in Fig. 3). We refer to the parent–child relationship between transactions in the hierarchy as a *nesting dependency*, which is noted as $T_{parent} \rightarrow T_{child}$. As an example, Fig. 3 shows a nesting dependency $T_1 \rightarrow T_2$. The nesting dependency is formally defined as follows:

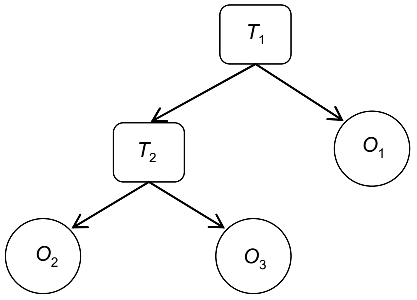


Fig. 3. An example of a transaction hierarchy in the business activity model: $\rightarrow$ denotes a nesting dependency. A transaction T_2 is composed of operations O_2 and O_3, and is nested within another transaction T_1.

Definition 1 (*Nesting dependency*)**.** Let us assume that there are two transactions T_p and T_c. The nesting dependency $T_p \rightarrow T_c$ occurs if T_c is executed as an operation of and executed from inside the scope of T_p.

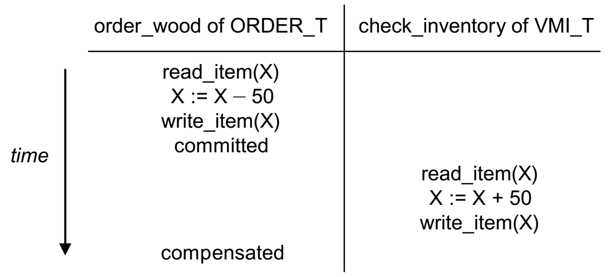
In the above definition, we refer to T_p as a *parent transaction* and T_c as a *child transaction*. A parent transaction is suspended until all its nested transactions terminate. That is, if any child transaction is working, the parent transaction cannot close itself. A child transaction appears atomic to its parent (i.e. it is completely executed or it is not executed at all).

Each transaction has a set of states {*active*, *complete*,[3] *cancel*, *compensate*, *end*}. An operation may also follow this state model since the operation can be executed as a transaction within its Web site.

- *active* is a state in which a transaction is working with its resources. Other transactions cannot access those resources.
- *complete* is a state in which a transaction has completed its work and released its resources.
- *cancel* is a state in which the work being done by a transaction has been cancelled.
- *compensate* is a state in which the work completed by a transaction has been compensated. The compensation semantically undoes the effects of the completed operations [3].
- *end* is a state in which the parent of a transaction has finalized its coordination successfully and thus the transaction has been closed.

Each transaction also has a set of possible state transitions among those states. When a transaction T is initiated, it enters the state *active*. T enters *complete* and release its resources before other transactions in the same parent are ready to complete, which relaxes the isolation property. If T *or* T's parent fails, its state is changed to *cancel*. The state of T becomes *end* if T's parent is ended. If T is a root,

[3] *Complete* and *cancel* are the terms used in the document of WS transaction specifications and correspond to well-known transaction terms, commit and abort, respectively.



	order_wood of ORDER_T	check_inventory of VMI_T

time	read_item(X)
	X := X – 50
	write_item(X)
	committed	

		read_item(X)
		X := X + 50
		write_item(X)

	compensated	

Fig. 4. The order_wood operation changes the value of X back to its previous value during its compensation; meanwhile, the check_inventory operation reads the temporary value of X.

T unilaterally enters the *end* state. If T's parent aborts or is compensated after T is completed, then its state is changed to *compensate*. A parent T_p can select which children are involved in overall outcome processing. T_p enters the *end* state after all the involved children are completed.

4.2. Inconsistency due to isolation relaxation

The inconsistency in the isolation-relaxing transaction model occurs when one transaction updates a data item and then the transaction fails. The updated item has been accessed by the other transaction before it is changed back to its origin value. We concretely illustrate the inconsistency problem by referring to the wood distribution scenario in Section 2. Let us assume that the data item X is the stock level of the inventory. Fig. 4 shows the situation in which the inconsistency occurs. The operation order_wood of the transaction ORDER_T updates the item X and then is committed according to the isolation-relaxation policy. Assume that another operation order_steel of ORDER_T fails. Thus, order_wood is compensated to change X back to its original value. Before the compensation is done, however, the transaction VMI_T reads the temporary value of X, which will not be recorded permanently in the database because of the failure of ORDER_T. After these operations are ended, VMI_T may produce the incorrect outcome derived from the incorrect value. Also, the data item X remains in an inconsistent state. To avoid this problem, VMI_T needs to be cancelled if it has not be committed or compensated if it has committed.

4.3. End-state dependency

We now introduce end-state dependency between transactions. As shown in the previous section, a transaction may need to be cancelled or compensated according to the final status of other transactions. That is, the successful *end* of the transaction is dependent upon the status of other transactions. The end-state dependency is defined as follows:

Definition 2 (*End-state dependency*). Let us assume that there are two transactions T_a and T_b. The end-state dependency $T_b \Rightarrow T_a$ exists if entering the *end* state of a transaction T_b is dependent upon the status of a transaction T_a.

If one transaction T_b reads a data item which has been updated by the other transaction T_a, the successful *end* of T_b is dependent upon the status of a transaction T_a. We formally define this case as follows:

Definition 3 (*Occurrence of end-state dependency*). Let us assume that there are two transactions T_a and T_b. The end-state dependency $T_b \Rightarrow T_a$ occurs if O_j reads a data item updated by O_i where $O_i \in T_a$ and $O_j \in T_b$.

In Definition 2, we refer to T_a as a *dominant transaction* and T_b as a *dependent transaction*. We also refer to O_i as a *dominant operation* and O_j as a *dependent operation*. A dominant transaction determines whether its dependent transactions should be successfully ended or not. For instance, T_b should be cancelled if T_a fails, because the update on the data referenced by T_b has been cancelled.

A nesting dependency also incurs an end-state dependency. A parent transaction T_p is dependent on its child transaction T_c in that T_p should be cancelled If T_c fails. Thus, we interpret a nesting dependency $T_p \rightarrow T_c$ as the end-state dependency $T_p \Rightarrow T_c$.

4.4. End-state dependency handling

When a dominant transaction fails, there are two solutions, *redo* and *abort*, to handle its dependent transactions. The redo scheme re-executes a dependent transaction. This scheme first identifies the operations affected by the cancellation of data updates. It then compensates or cancels the identified operations depending on whether each operation has been completed or is running. Finally, it executes these operations again. Let us consider the example in Fig. 5. Two transactions, T_1 and T_2, have an end-state dependency $T_2 \Rightarrow T_1$, which occurs because an operation O_{22} of T_2 has read the value updated by the operations O_{11} of T_1. The operations, O_{21}, O_{22}, and O_{11} (colored by gray), have already been completed. We now assume that O_{12} fails and thereby O_{11} is compensated for ensuring the atomicity of T_1. The compensation nullifies the effect of the update of O_{11}. In this case, the redo scheme identifies O_{22} as an affected operation because it has read the cancelled update. The scheme also

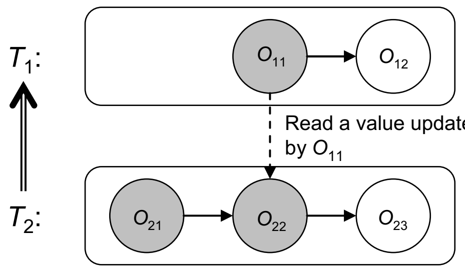


Fig. 5. An example of dependencies: $\Rightarrow$ denotes an end-state dependency and the arrows between operations denote precedence in an execution order of operations. Gray-colored circles indicate completed operations.

identifies O_{23}, which is a subsequent operation of the dependent operations O_{22}. This is because O_{23} may have used the old output of O_{22}. Thus, O_{23} needs to cancel its execution and re-execute itself with the new output of O_{22}.

On the other hand, the abort scheme is to stop the execution of a dependent transaction and undo its data updates. The undo is done by compensating all completed operations and canceling all running operations. When applied to the above example, the scheme cancels O_{23} and compensates O_{21} and O_{22}.

Both schemes require some operations to be compensated. This compensation may recursively abort succeeding dependent transactions, which could result in a lot of transaction aborts. The redo scheme has lower possibility of the recursive compensation than the abort scheme. This is because, as mentioned above, the redo scheme needs to compensate only dependent operations and their following operations. In contrast, the abort scheme compensates all completed operations. In the example above, O_{21}, which is not compensated in the redo, is compensated in the abort. Hence, if applications allow re-executions of transactions, the redo scheme is more desirable.

As described so far, dependent transactions may need to be re-executed or aborted due to the failures of their dominant transactions. Thus, dependent transactions should delay entering the end state until their dominant transactions are all successfully ended.

4.5. Circular waiting

The delayed ending of dependent transactions can incur a circular waiting state, in which each transaction cannot enter the *end* state even though it has been completed. We formally define circular waiting as follows:

Definition 4 (*Circular waiting*). A set of transactions is in a state of circular waiting (1) if all transactions in the set are completed and (2) each transaction T_i in the set has an end-state dependency $T_i \Rightarrow T_j$ where T_j is another transaction in the set.

In Definition 4, we refer to the transaction set as a *waiting set*. As an example of circular waiting, let us assume that two transactions T_1 and T_2 have end-state dependencies in both directions, i.e. $T_1 \Rightarrow T_2$ and $T_2 \Rightarrow T_1$. After completing all their operations, T_1 and T_2 will be in a state of circular waiting; T_1 is waiting for T_2 to be ended, while T_2 is waiting for T_1 to be ended. Both T_1 and T_2 cannot be ended themselves. Meanwhile, the applications, which have issued T_1 and T_2, cannot proceed to their next tasks. Circular waiting is also possible when more than two transactions are involved (e.g. $T_1 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_1$).

To solve this problem, we need to be able to detect circular waiting. We use a data structure called *End-Wait-For-Graph* (EWFG) which is defined as follows:

Definition 5 (*End-Wait-For-Graph*). A EWFG $\langle N, E\rangle$ is a directed graph, where a vertex i in N represents a completed transaction T_i and an edge in E from vertex i to vertex j indicates an end-state dependency $T_i \Rightarrow T_j$.

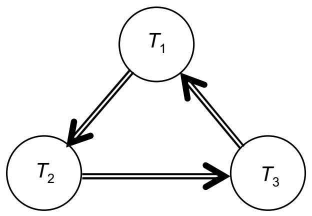


Fig. 6. A End-Wait-For-Graph which shows the state of circular waiting.

Fig. 6 shows an example of EWFG, in which transactions T_1, T_2, and T_3 are completed and has end-state dependencies $T_1 \Rightarrow T_2$, $T_2 \Rightarrow T_3$, and $T_3 \Rightarrow T_1$, respectively. We say that a set of transactions is in a state of circular waiting if and only if its EWFG has one or more cycles. Thus, we need to detect cycles in EWFG to know whether circular waiting exists. This detection approach needs to build and maintain EWFG across multiple transactions. Section 5.4 describes the algorithm to detect circular waiting in a distributed Web services environment.

If a cycle is discovered, each transaction in the waiting set knows that all other transactions in the same set have been competed. Thus, the transaction can enter the *end* state without worrying that it will be re-executed or aborted.

5. Web services transaction dependency management protocol

5.1. Overview

In this section, we propose a new protocol called Web services Transaction Dependency management Protocol (WTDP). Our design goals of WTDP are to ensure interoperability and security, which are common and core requirements in Web services environments. Interoperability here means that WTDP-enabled coordinators and participants can seamlessly cooperate with existing ones not supporting WTDP. For this, WTDP is designed as an extension of the WS-BusinessActivity protocol, not requiring any modifications on the protocol. WTDP adds new services and their related messages which do not conflict with the existing protocols. Maintaining security is crucial in business trans-

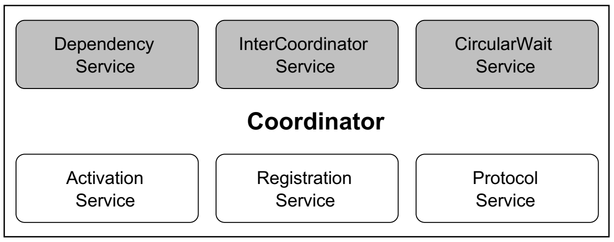


Fig. 7. Services at a coordinator: Gray-colored ones are the services provided by WTDP.

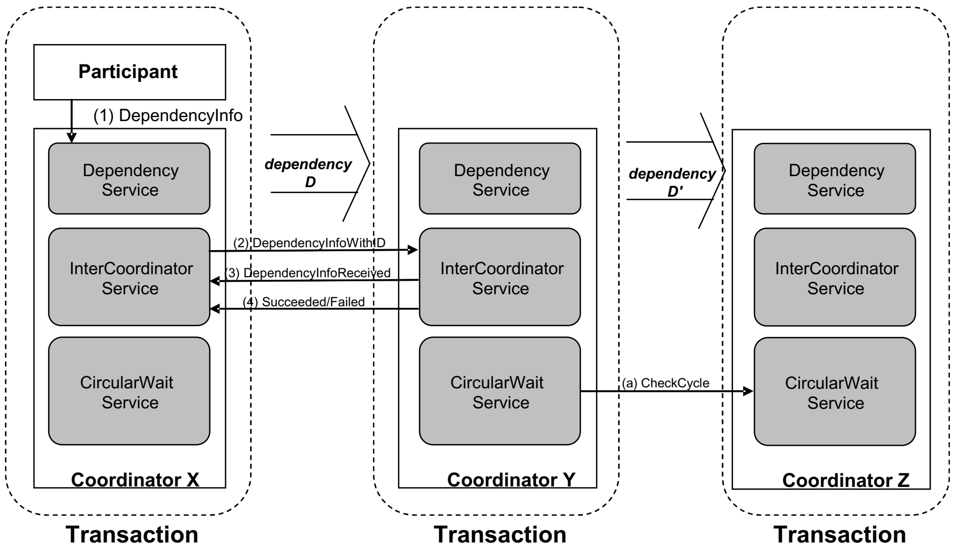


Fig. 8. The message flow of WTDP. There are two end-state dependencies – D and D′. D is between transactions managed by coordinators X and Y. D′ is between transactions of coordinators Y and Z.

actions especially among different, loosely-coupled organizations. End-state dependencies can be interpreted as mission-critical information such as confidential contracts between organizations. WTDP makes such information to be stored at distributed coordinators, in which each coordinator manages only its involving end-state dependencies. This policy prevents a coordinator to steal a look at other organizations' dependencies.

WTDP consists of three services, which are added to coordinators as shown in Fig. 7. *Dependency service* is used by participants to inform their coordinators of dependency information whenever participants detect end-state dependencies. *InterCoordinator service* is used to exchange dependency information among coordinators. Also, the ending status of dominant transactions such as failure or success is communicated among the coordinators of the transactions through an InterCoordinator service. *CircularWait service* is used to detect and resolve circular waiting. A token-based approach is used to handle circular waiting over dependency information distributed over multiple coordinators. These services are defined in WSDL.[4] Each service uses one or more messages to communicate with the services of other coordinators.

Fig. 8 shows WTDP messages and their flows. When an end-state dependency *D* is detected at a participant site, it informs a dependent coordinator[5] of *D* by sending a `DependencyInfo` message (1). The dependent coordinator informs its dominant coordinator by using a `DependencyInfoWithID` message (2). The dominant coordinator acknowledges the message by returning a `DependencyInfoReceived` message (3). Now, the dominant coordinator sends `Succeeded` or `Failed` messages to the dependent coordinator according to the ending status of the dominant transaction (4). The dependent coordinator handles its transaction appropriately according to the ending status of the dominant transaction.

Meanwhile, the dominant transaction may additionally have an end-state dependency D' with another transaction. In this case, the dependency relations may become a cycle. To determine if there is circular waiting, the dominant coordinator of *D* sends a `CheckCycle` message to the dominant coordinator of D' (a). If a cycle exists, the `CheckCycle` message will finally be returned to the sender. This detection process is performed periodically. Detailed description of the messages and their flows are described in subsequent subsections.

5.2. Dependency service

A participant detects end-state dependencies by observing activities of operations running on itself. When a participant detects a dependent operation (which reads a value updated by another operation), it notifies the coordinator of the dependent operation that the end-state dependency has occurred. This notification is accomplished by sending a `DependencyInfo` message to the Dependency service of the dependent coordinator. The Dependency service is defined as Fig. 9. A `DependencyInfo` message contains dependency information consisting of IDs of

```
<wsdl:portType name="DependencyPortType">
  <wsdl:operation name="InformDependency">
    <wsdl:input message="dep:DependencyInfo"/>
  </wsdl:operation>
</wsdl:portType>
```

Fig. 9. Dependency service.

[4] Web Services Description Language (WSDL) is a standard language for describing the operations and messages of Web services.

[5] Throughout this paper, we refer to a coordinator instance managing a dependent transaction as a *dependent coordinator* Likewise, we use a term called *dominant coordinator*.

Table 1
Elements in a `DependencyInfo` message

Element	Description
domTID	Dominant transaction's ID
domOID	Dominant operation's ID
domInterCoordinatorAddr	Address of a InterCoordinator service in a dominant coordinator
depTID	Dependent transaction's ID
depOID	Dependent operation's ID

dominant and dependent transactions in an end-state dependency relation (see Table 1). The message also includes the address of an InterCoordinator service in a dominant coordinator, to which the dependent coordinator will deliver a dependency ID and the address of its InterCoordinator service (see the subsequent section).

5.3. InterCoordinator service

Four kinds of messages, `DependencyInfoWithID`, `DependencyInfoReceived`, `Succeeded`, and `Failed`, are transferred between InterCoordinator services of dominant and dependent coordinators (see Fig. 10). The dependent coordinator uses `DependencyInfoWithID` to carry a dependency ID to its dominant coordinator (see Table 2). The dependency ID will be used by a dominant coordinator to notify the dependent coordinator which end-state dependency is affected due to the end of the dominant transaction. The dominant coordinator replies to the dependent coordinator with a `DependencyInfoReceived` message, which is an acknowledgement that the `DependencyInfoWithID` message has been received. The `DependencyInfoReceived` message also includes the address of the CircularWait service of the dominant coordinator as shown in Table 3. The dependent coordinator will send messages to this address in order to detect circular waiting.

`Succeeded` and `Failed` messages are used by a dominant coordinator to inform its dependent coordinator of the ending status. If the dependent coordinator receives a `Succeeded` message, it ends its dependent transaction. On the other hand, if the dependent coordinator receives a `Failed` message, it should abort or re-execute its dependent transaction as mentioned in Section 4.3. `Succeeded` and `Failed` messages include the same set of elements (see

Table 2
Elements in a `DependencyInfoWithID` message

Element	Description
domTID	Dominant transaction's ID
domOID	Dominant operation's ID
depTID	Dependent transaction's ID
depOID	Dependent operation's ID
depInterCoordinatorAddr	Address of a InterCoordinator service in a dependent coordinator
dependencyID	ID of an end-state dependency

Table 3
An element in a `DependencyInfoReceived` message

Element	Description
domCircularWaitAddr	Address of a CircularWait service in a dominant coordinator

Table 4
Elements in `Succeeded` and `Failed` messages

Element	Description
domTID	Dominant transaction's ID
dependencyID	ID of an end-state dependency

Table 4). The dependencyID element indicates which end-state dependency is affected by the end of a dominant transaction.

5.4. CircularWait service

If a dominant transaction is further dependent on other transactions, there is a possibility that the dependency relations form a cycle. As mentioned in Section 4.5, the cycle among completed transactions means that circular waiting occurs. The easiest way to detect cycles would be maintaining information on all dependencies at a central single node. We can easily detect cycles from consolidated information stored in the same memory space. However, this approach causes security problems. End-state dependencies can be interpreted as mission-critical information. Thus, it is better to manage this information at distributed points. We propose a token-based scheme that detects cycles from end-state dependency information distributed over multiple coordinators. The basic idea of this scheme is that we

```
<wsdl:portType  name="InterCoordiPortType">
      <wsdl:operation name="TransmitDependencyInfo">
            <wsdl:input message="intercoordi:DependencyInfoWithID"/>
            <wsdl:output message="intercoordi:DependencyInfoReceived"/>
      </wsdl:operation>
      <wsdl:operation name="InformSucceeded">
            <wsdl:input message="intercoordi:Succeeded"/>
      </wsdl:operation>
      <wsdl:operation name="InformFailed">
            <wsdl:input message="intercoordi:Failed"/>
      </wsdl:operation>
</wsdl:portType>
```

Fig. 10. InterCoordinator service.

```
<wsdl:portType  name="CircularWaitPortType">
      <wsdl:operation name="SendCheckCycle">
            <wsdl:input message="circularwait:CheckCycle"/>
      </wsdl:operation>
</wsdl:portType>
```

Fig. 11. CircularWait service.

Table 5
Elements in a `CheckCycle` message

Element	Description
initiatorTokenID	Token identifier that is assigned by its creator.

can recognize a cycle when a coordinator receives the token that was initiated by itself.

A CircularWait service uses the `CheckCycle` message to execute the token-based scheme (see Fig. 11). When a dominant coordinator receives a `DependencyInfoWithID` message, it initiates a new token with a unique ID (i.e. initiatorTokenID in Table 5). Then, the dominant coordinator sends the token to its every dominant coordinator. The token is recursively traversed along dependency relations. If the initiating coordinator receives the token initiated by itself, there must be a circular dependency. Note that circular waiting occurs in a set of completed transactions. Thus, each coordinator delivers the token to the dominant coordinators only when its transaction is completed. The detailed algorithm for the detection scheme is given below. We also give the correctness proof for the algorithm.

```
For the coordinator $C_i$ initiating the token-based
  detection for its dominant transaction $T_i$:
if $T_i$ is completed
then
  for all coordinators $C_j$ such that $C_j$ is a dominant
    coordinator of $C_i$
    send a CheckCycle(id) message to $C_j$ where id
      is set as initiatorTokenID

For the coordinator $C_i$ of a dominant transaction $T_i$ on
  receiving CheckCycle(id):
if id = initiatorTokenID of $T_i$
then
  declare that $T_i$ is deadlocked
else
  if $T_i$ is completed
  then
    for all coordinators $C_j$ such that $C_j$ is a dominant
      coordinator of $C_i$
      send a CheckCycle(id) message to $C_j$
```

Theorem 1. *If the coordinator C_i is on a cycle in EWFG when it initiates the detection process then it will eventually get its initiating* `CheckCycle` *message.*

Proof. Let C_i be on a cycle. C_i will send a `CheckCycle` message to its successor vertex C_j (i.e. the dominant coordinator of C_i). Similarly, C_j will send the `CheckCycle` message to its dominant coordinator. Thus, C_i will eventually receive the message. □

Theorem 2. *If the coordinator C_i receives its initiating* `CheckCycle` *message then it is on a cycle in EWFG when this message is received.*

Proof. The coordinator C_i is an initiator of the detection process and thereby is the only vertex which can send a `CheckCycle` message without having received the message. Hence if the initiator C_i receives `CheckCycle`, there exists a finite sequence $C_{j(0)}, \ldots, C_{j(n)}$ where 1) $C_{j(0)} = C_{j(n)} = C_i$ and 2) $C_{j(k)}$ received `CheckCycle` from $C_{j(k-1)}$ at time t_k, and $t(k-1) < t(k), k = 1 \ldots n-1$. □

6. Prototype implementation

To validate the proposed protocol, we implemented a prototype system which supports WTDP. It consists of a coordinator module and a participant module. All the WTDP services are implemented. The prototype also supports most functions specified in the WS-Coordination and the WS-BusinessActivity specifications. We develop the prototype on the Linux operating system using Java. Both modules use Apache Tomcat 4.1.18 and Apache Axis 1.1 beta as a Web server and a SOAP engine, respectively.

An important issue in implementing the prototype is the automatic detection of end-state dependencies at participant sites. In order to find end-state dependencies between two operations on Web services, participants should continuously monitor which data items[6] are used or updated by the operations. If an operation uses any data updated by another operation before the transaction including the latter operation is ended, an end-state dependency occurs between the two operations. However, supervising data accesses and their contexts enforces the modification of database systems, which is very complicated in practice. For rapid prototyping, we identify dependencies by only monitoring execution sequences of operations.

This WS-operation-level detection scheme may incur false detections – an end-state dependency may be detected between two operations even when there is no end-state dependency. Let us assume that a data is changed by an operation O_A and then read by another operation O_B. Sometimes, O_A may not change the value of the data. In this case, since the participant hosting both operations does not know that the data is not changed, it will conclude that an end-state dependency happens. As a result, the transaction including O_B unnecessarily delays its ending, waiting for the end of the transaction including O_A. Note that these

[6] Note that we assume the type of participant's data sources is a database.

delays do not incur any inconsistent outcome of transactions. Also, the delays do not reduce resource utilization because the waiting transactions have already been completed and released the resources.

In this section, we first describe the architecture of the participant and the coordinator modules in Sections 6.1 and 6.2, respectively. We then discuss how to identify end-state dependencies in our prototype in Section 6.3.

6.1. Participant module

The participant module is located at each organization's Web services system and manages the operations on Web services. It detects end-state dependencies and notifies its coordinator of them. Fig. 12 shows the architecture of the participant module, which is composed of *Message Handler* (MH), *Web services engine*, *Application Container* (AC), *Local Compensator* (LC), *Operation Manager* (OM), and *End-state Dependency Detector* (EDD). Following the WS Transactions specifications, the Activation Service, the Registration Service, and the Protocol Service are also implemented. The followings are brief descriptions on components.

- *Message Handler* is embedded in the Web services engine, inspects all incoming and outgoing messages, and forwards the messages to appropriate components. For instance, requests to service logics are forwarded to AC while messages from Coordinators are forwarded to OM or relevant service handlers.
- *Application Container* hosts service logics in the form of Web services. Upon each request to a Web services, it not only invokes the service routine but sends summary information on the invoked Web services to EDD.
- *Local Compensator* takes charge of automatic compensation for already completed operations. For that, it holds compensation information of operations until their transactions are closed. Upon a compensation request, it undoes the completed operation using stored compensation information.
- *Operation Manager* handles operations. Whenever it receives a CoordinationContext – which represents the information on a transaction being created – from a service requestor (i.e. an application), it creates an object for the transaction.
- *End-state Dependency Detector* monitors the occurrence of end-state dependencies among operations. Whenever an operation on a Web service is invoked, EDD checks whether the operation incurs any dependencies on other running operations. The detailed description on end-state dependency detection is discussed in Section 6.3.

6.2. Coordinator

The coordinator is a core module to manage transactions with the helps of relevant participants. Also, it detects and resolves the circular waiting status. The architecture of the coordinator is shown in Fig. 13.

- *Transaction Manager* (TM) creates and manages transactions. Interested readers can refer to WS transactions Specifications [10].
- *Dependency List* (DL) keeps and manages all the dependency information.
- *Dependency Manager* (DM) exchanges end-state dependency information with those in other coordinators through an InterCoordinator service and store the information in DL. It also notifies dependent coordinators whether their dominant transaction is failed or ended successfully. DM is designed to have only the information of the transactions managed by its coordinator and their dominant or dependent transactions. It is prohibited to access the information of any other transactions. Although restricting the access of other transaction's information makes circular wait resolving more complicated, it makes the mechanism to be more secure.

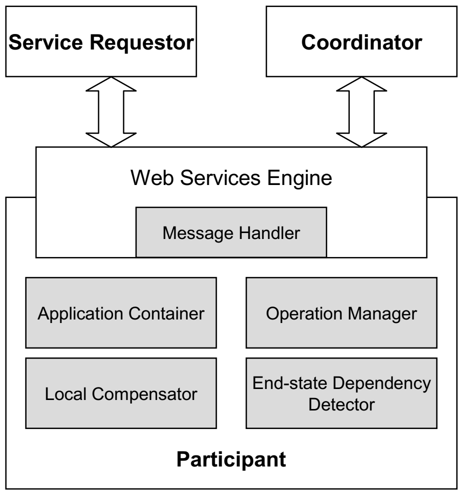


Fig. 12. Architecture of the participant module.

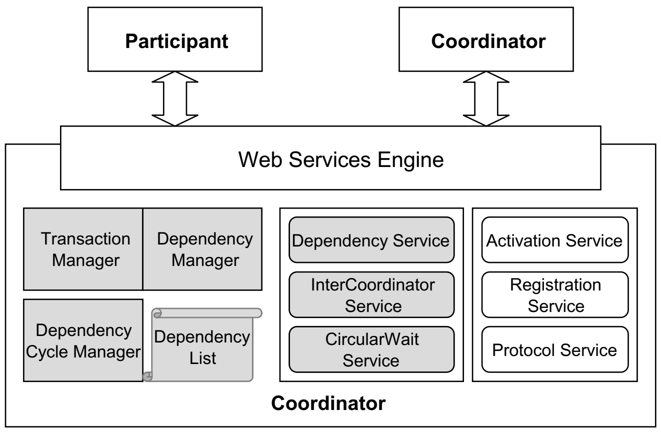


Fig. 13. Architecture of the coordinator.

- *Dependency Cycle Manager* (DCM) checks and resolves the circular waiting status. It sends a token to its dominant coordinator and receives a response to the token. Based on the response, it identifies the existence of a circular dependency.

6.3. End-state dependency detection mechanism

Detecting end-state dependencies is enabled by specifying relations among WS operations accessing same data items. This specification may be done by application developers or business process designers. They know when end-state dependencies may happen better than anyone else. For our motivating scenario, we have specified the relation between the order wood and the check inventory operations since both of them access the data 'total stock'. The whole detection process can be divided into three phases: *description*, *registration*, and *discovery*. Fig. 14 shows the three phases. Note that participants need an additional module, EDD. It takes charge of detecting and reporting the occurrence of end-state dependencies using the relation among operations.

The first phase, description, is supposed to be executed before a participant is in service. In the description phase, Web services developers provide the dependency information. We describe it in the next paragraph in detail. The second phase, registration, is done when the participant starts the published Web services. The dependency information is loaded into EDD. In the third phase, discovery, whenever the Web services call is served by the participant, it checks whether there is an end-state dependency among the transactions. The followings are detailed description of the phases.

To ease the description phase, we used a simple formal language called *end-state dependency description language* (EDDL), which is built on XML. It represents a relationship between a dominant operation and a dependent operation. In Fig. 15, a simple example of a dependency description is shown. Table 6 describes elements of EDDL.

The second phase, registration, takes places when participant's Web services engine starts to operate. In this phase, the described information is loaded into EDD. The dependency information is converted into mapping between dominant operations and dependent operations, which is in turn written into a *dependency map* (DMAP). DMAP is a data structure that maintains the mapping information.

The third phase, discovery, is performed repetitively upon each call to operations. Every time a Web service operation is called, the participant module notifies EDD of the information on the call. Based on DMAP and the *Candidate LIST of Dominant Operation Calls*

Table 6
Elements of EDDL and descriptions

Element	Description
domOP	Dominant operation which incurs the end-state dependency
name	Operation name
depOP	Dependent operation which has an effect of the end-state dependency

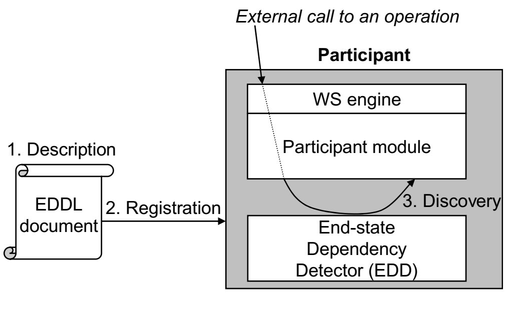


Fig. 14. Three phases of detection in the prototype.

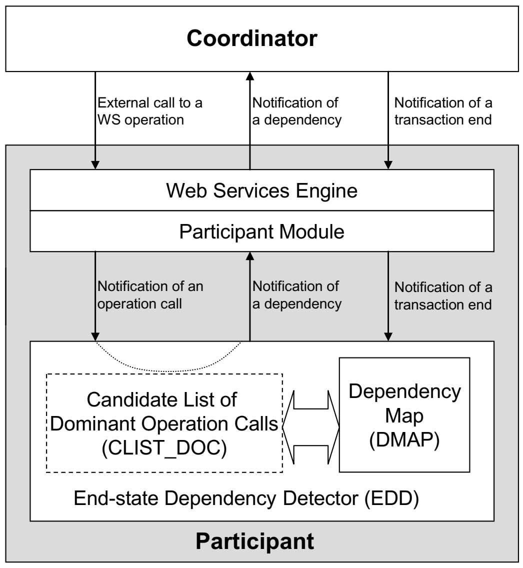
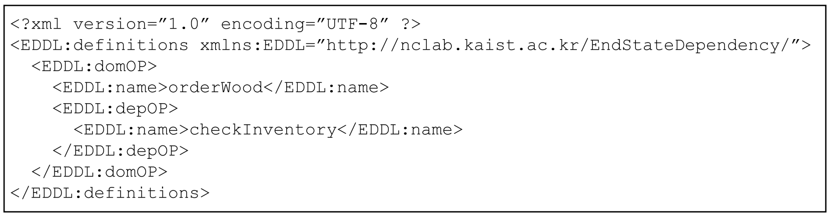

Coordinator
External call to a WS operation
Notification of a dependency
Notification of a transaction end
Web Services Engine
Participant Module
Notification of an operation call
Notification of a dependency
Notification of a transaction end
Candidate List of Dominant Operation Calls (CLIST_DOC)
Dependency Map (DMAP)
End-state Dependency Detector (EDD)
Participant

Fig. 16. Discovery phase.



```
<?xml version="1.0" encoding="UTF-8" ?>
<EDDL:definitions xmlns:EDDL="http://nclab.kaist.ac.kr/EndStateDependency/">
  <EDDL:domOP>
    <EDDL:name>orderWood</EDDL:name>
    <EDDL:depOP>
      <EDDL:name>checkInventory</EDDL:name>
    </EDDL:depOP>
  </EDDL:domOP>
</EDDL:definitions>
```

Fig. 15. An example of EDDL: Operation *orderWood* is the dominant operation, and the *checkInventory* is the mapped dependent operation.

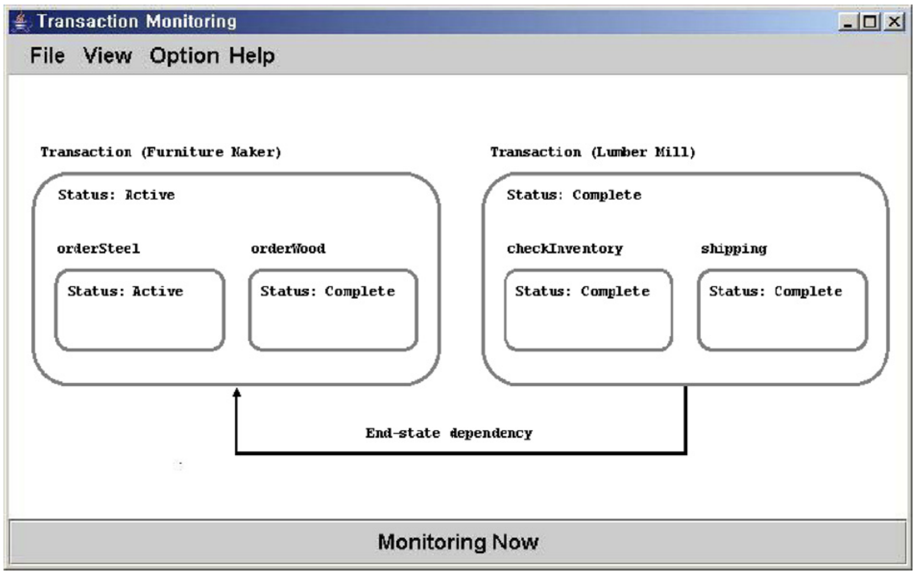


Fig. 17. A screen shot of the monitoring utility of the prototype system: Two large boxes mean two transactions; the furniture maker initiates the left transaction and the lumber mill initiates the right transaction. Although the right transaction has finished all its child transactions, it still waits for the closure of the left transaction.

(CLIST_DOC), EDD can find an end-state dependency and notifies its coordinator of the dependency. CLIST_DOC maintains the list of active dominant-operations which are completed but still can be compensated later since their transactions are not ended successfully yet. The call information of an operation will be deleted from CLIST_DOC when the transaction including the operation is closed. Fig. 16 shows this discovery phase.

Using the WTDP prototype, we implemented the scenario described in Section 2. Each company runs different Web services management systems on its own machine. In addition, different coordinators are assigned to different companies. All the Web services for the scenario and Web pages for user interface are implemented. The dependency between order transaction and VMI transaction is described by EDDL and loaded into the Wood distributor's EDD before running the scenario. The scenario runs correctly and the end-state dependency is identified. Also, its information is forwarded to relevant coordinators once it happens. Fig. 17 shows a screen shot of our monitoring utility which is a part of our WTDP prototype. It displays the current status of end-state dependencies of a coordinator using the relevant information stored in that coordinator. In the figure, a transaction initiated by the furniture maker, order transaction, and a transaction initiated by the lumber mill, VMI transaction, are displayed along with the end-state dependency between them.

7. Conclusions

The isolation property of WS transactions is usually relaxed for a high level of concurrency. Relaxed isolation brought up the notion of end-state dependency among transactions; when a transaction accesses intermediary results of another transaction, its successful end depends on the ending status of the other. In this paper, we proposed a mechanism which guarantees the consistent executions of isolation-relaxing WS transactions. The mechanism effectively detects inconsistent states of transactions and recovers them to consistent states. We also proposed a dependency management protocol called WTDP. By employing the WTDP, organizations save time and money in managing end-state dependencies. Since WTDP is designed to incorporate with the WS transactions specifications, it is easily deployed in existing WS-transactions-compliant transaction management systems.

We implemented a prototype system which supports WTDP. The biggest problem encountered during the implementation was the difficulty of constructing the automatic-detection feature of end-state dependencies on existing database systems. To fully support this feature, the modification of the database systems is required, which is very complicated in practice. Rather, we identify dependencies by only monitoring execution sequences of operations. This approach may incur false detection because we do not have the information on reads and writes executed by each operation at run time. Note that the false detection does not incur any inconsistent outcome of transactions and does not reduce resource utilization.

References

[1] R. Akkiraju, D. Flaxer, H. Chang, T. Chao, L.-J. Zhang, F. Wu, J.-J. Jeng, A framework for facilitating dynamic e-business via web services, in: Proceedings of OOPLSA 2001 Workshop on Object-Oriented Web Services, Florida, USA, 2001.

[2] Arjuna Technologies Ltd., Fujitsu Software, IONA Technologies PLC, Oracle Corp, and Sun Microsystems, 2003. Web Services Composite Application Framework (WS-CAF). Available from <http://developers.sun.com/techtopics/webservices/wscaf>.

[3] D. Biswas, Compensation in the world of web services composition, in: Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, Revised Selected Papers, vol. 3387 Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 69–80.

[4] F. Curbera, W.A. Nagy, S. Weerawarana. Web services: Why and How, in: Proceedings of OOPLSA 2001 Workshop on Object-Oriented Web Services, Florida, USA, 2001.

[5] S. Dalai, S. Temel, M. Little, M. Potts, J. Webber, Coordinating business transactions on the web, IEEE Internet Computing 7 (1) (2003) 30–39.

[6] J. Eliot, B. Moss Nested Transactions: An Approach to Reliable Distributed Computing, MIT Press, Cambridge MA, 1985.

[7] K. Elmagarmid, Y. Leu, W. Litwin, M. Rusinkiewicz. A multi-database transaction model for interbase, in: Proceedings of the 16th VLDB Conference, 1990, pp. 507–518.

[8] H. Garcia-Molina, K. Salem, SAGAS, in: Proceedings of ACM SIGMOD Conference, 1987, pp. 249–259.

[9] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufman Publishers, 1993.

[10] IBM, Microsoft, and BEA, Web services transactions specifications. Available from: <http://www-128.ibm.com/developerworks/library/specification/ws-tx/>, 2005.

[11] S. Mehrotra, R. Rastogi, H.F. Korth, A. Silberschatz, A transaction model for multi-database systems, in: Proceedings of the 12th International Conference on Distributed Systems, 1992, pp. 56–63.

[12] T. Mikalsen, S. Tai, I. Rouvellou. Transactional attitudes: Reliable composition of autonomous Web services, in: DSN 2002, Workshop on Dependable Middleware-based Systems, 2002.

[13] S. Narayanan, S.A. Mcllraith, Simulation, verification and automated composition of web services, in: Proceedings of WWW Conference, Honolulu, Hawaii, USA, 2002.

[14] M.P. Papazoglou, Web Services and Business Transactions, World Wide Web Journal 6 (2003) 49–91.

[15] M. Pierce, C. Youn, G. Fox, S. Mock, K. Mueller, O. Balsoy. Interoperable web services for computational portals, in: Proceedings of the IEEE/ACM SC2002 Conference, Baltimore, USA, 2002, pp. 16–22.

[16] OASIS, Business transaction protocol, Available from: <http://www.oasis-open.org/committees/documents.php?wg_abbrev=business-transaction>, 2005.

[17] B.A. Schmit, S. Dustdar. Towards transactional web services, in: Proceedings of 1st IEEE International Workshop on Service-oriented Solutions for Cooperative Organizations, Munich, Germany, 2005.

[18] S. Tsur, Are web services the next revolution in E-commerce?, in: Proceedings of the 27th VLDB Conference, Roma, Italy, 2001.

[19] G. Weikum, H.J. Schek, Concepts and Applications of Multilevel Transactions and Open Nested Transactions, in: A. Elmagarmid (Ed.), Database Transaction Models for Advanced Applications, Morgan Kaufman Publishers, 1992, pp. 515–553.

[20] Definition of Vendor Managed Inventory, Available from: <http://www.vendormanagedinventory.com/definition.htm>.

[21] Zhang, M. Nodine, B. Bhargava, O. Bukhres. Ensuring Relaxed Atomicity for flexible transactions in multi-database systems, in: Proceedings of ACM SIGMOD Conference, 1994, pp. 67–78.