Maintaining Consistency Under Isolation Relaxation of Web Services Transactions

Seunglak Choi[1], Hyukjae Jang[2], Hangkyu Kim[1],
Jungsook Kim[3], Su Myeon Kim[2], Junehwa Song[2], and Yoon-Joon Lee[1]

[1] 371-1 Guseong-Dong, Yuseong-Gu, Daejeon, Korea
{slchoi, hkkim, yjlee}@dbserver.kaist.ac.kr
[2] 371-1 Guseong-Dong, Yuseong-Gu, Daejeon, Korea
{hjjang, smkim, junesong}@nclab.kaist.ac.kr
[3] 161 Gajeong-dong, Yuseong-gu, Daejeon, Korea
jungsook96@etri.re.kr

Abstract. For efficiently managing Web Services (WS) transactions which are executed across multiple loosely-coupled autonomous organizations, isolation is commonly relaxed. A Web services 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 causes 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 a *completion 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.

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 [9,10,11,12,13]. 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 [6], Business Transaction Protocol (BTP) [7], and WS-CAF [8]. Also, commercial transaction management systems [17,18] have been developed implementing these protocols. For efficient

M. Kitsuregawa et al. (Eds.): WISE 2005, LNCS 3806, pp. 245 – 257, 2005.

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] [19]. In contrast to traditional transactions, WS transactions live long – *e.g.,* a few hours or days. Thus, 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 [1,2,5,6,7,8,14] 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 inconsistency 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 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 a *completion dependency*. The notion is a relationship between two transactions in which one transaction's failure incurs the inconsistent state of 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 completion until its related transactions are closed, *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 completed and thereby no one can be completed. The mechanism provides a graph-based method to detect circular waiting and defines a condition to safely complete 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 completion dependency. Upon its occurrence, using the protocol, participants automatically delivers 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 completion of related transactions when all the relevant dependencies are resolved. In addition, WTDP detects and resolves

[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.

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 can be 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. Finally, we conclude our work in Section 6.

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 [15]. Under the VMI contract, producers are responsible for managing and replenishing consumers' inventories. Thus, consumers can reduce the overhead of inventory management. 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 when the amount of wood in the inventory goes below an agreed level of stocks.

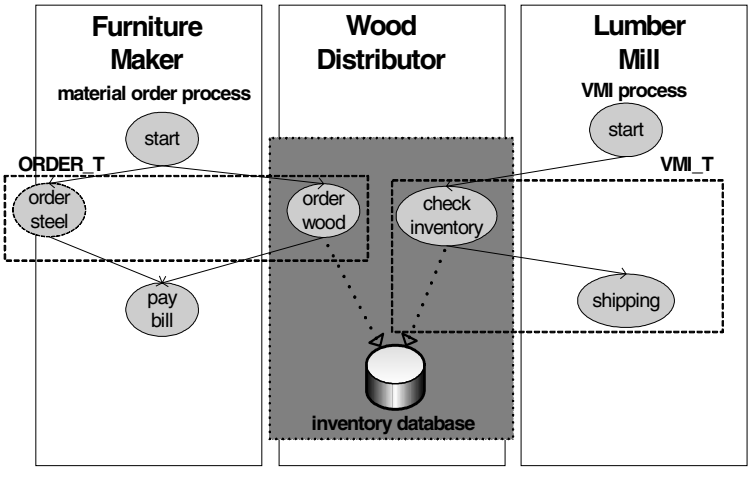

Fig. 1. The occurrence of dependency between two Web Services transactions

Figure 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 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's 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 [1] 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 [2] 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 [3,4,5] 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 [6] 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) [7] was proposed by the OASIS business transactions technical committee, and also relaxes isolation of transactions. WS-CAF [8], 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.

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.

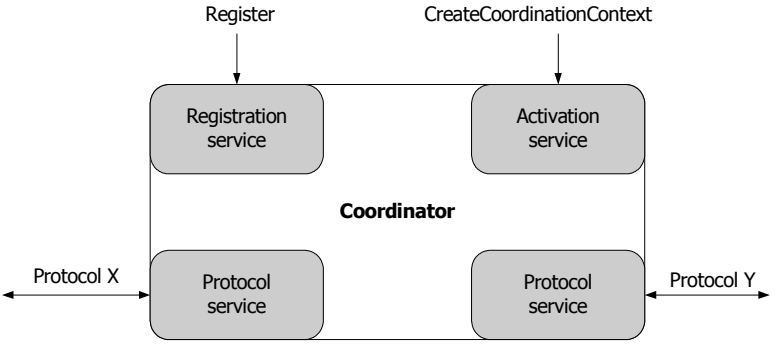

Fig. 2. Coordinator architecture

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 Figure 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.

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 Consistency Maintaining

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 formally define the completion dependency in Section 4.2, which is used to determine transactions affected by a given transaction's failure. In Section 4.3, we describe how to handle the dependency in Section 4.3. 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 resolve circular waiting in Section 4.4.

4.1 Underlying Transaction Model

Our mechanism is designed for the transactions following the business activity model of the WS-Transactions specifications. In that model, a transaction consists of multiple operations on a collection of Web services. The model relaxes the isolation property in that an operation can complete and release its resources before other operations in the same transaction are ready to complete. If a transaction fails before getting to the complete phase, to ensure the atomic execution of the transaction, the running operations are *cancelled*[3] and the completed ones are *compensated*. The compensation semantically undoes the effects of the completed operations.

4.2 Completion Dependency

We now introduce completion dependency between transactions. As shown in Section 2, a transaction may need to be cancelled according to the status of other transactions. The completion dependency is notated as $T_b \Rightarrow T_a$ when the successful completion of a transaction T_b is dependent upon the status of a transaction T_a. The completion dependency can be formally defined as follows:

Definition 1 (Completion dependency). Let us assume that there are two transactions T_a and T_b. The completion dependency $T_b \Rightarrow T_a$ occurs if O_j reads a value updated by O_i where $O_i \in T_a$ and $O_j \in T_b$.

In the above definition, 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.

4.3 Completion 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

[3] Cancellation and completion are the terms used in the document of WS transactions specifications and correspond to well-know transaction terms, abort and commit, respectively.

then compensates or cancels the identified operations depending on whether the operations has been completed or is running. Finally, it executes these operations again. Let us consider the example in Figure 3. Two transactions, T_1 and T_2, have a completion 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 identifies O_{23}, which is a subsequent operation of the dependent operations ST_{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 can 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 the completion until their dominant transactions are all successfully ended.

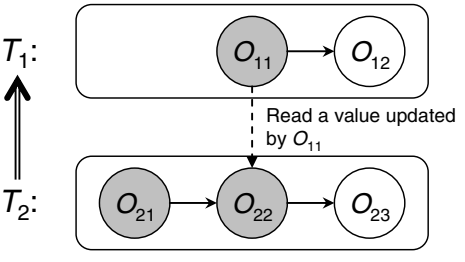

Fig. 3. An example of dependencies: ⇒ denotes a completion dependency and → denotes precedence in an execution order of operations. Gray-colored circles indicate completed operations.

4.4 Circular Waiting

The delayed completion of dependent transactions can incur circular waiting in which each of two transactions is waiting for the other to be ended. For example, let us assume that two transactions T_1 and T_2 have completion 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 completed 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.* $T_1 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_1$).

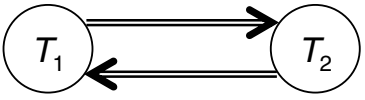

Fig. 4. A dependency graph which shows the state of circular waiting

To solve this problem, we need to be able to detect circular waiting. We construct a *dependency graph* where a node represents a transaction and an edge represents a completion dependency. Figure 4 shows the dependency graph for the example above. We say that a set of transactions is in a state of circular waiting if and only if its dependency graph has a cycle. This approach needs to build and maintain the dependency graphs across multiple Web services, which will be described in Section 5.4. If a cycle is discovered, the condition for the completions of the transactions in the cycle is changed. A transaction can complete if *all other transactions* in the same cycle are *ready* to complete. When such a condition is met, there is no possibility that transactions 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 transactions especially among different, loosely-coupled organizations. Completion dependencies can be interpreted as mission-critical information

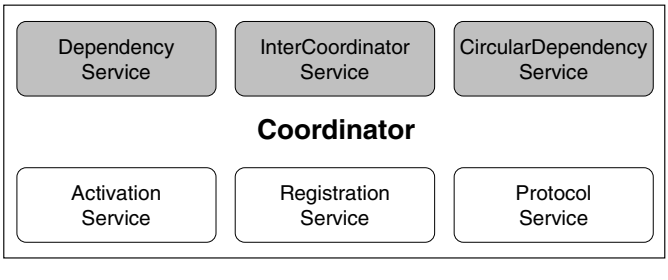

Fig. 5. Services at a coordinator: gray-colored ones are the services provided by WTDP

such as confidential contracts between organizations. WTDP makes such information to be stored at distributed coordinators, in which each coordinator manages only its involving completion 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 Figure 5. *Dependency service* is used by participants to inform their coordinators of dependency information whenever participants detect 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. *CircularDependency 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. [16] includes the definitions of all services and messages.

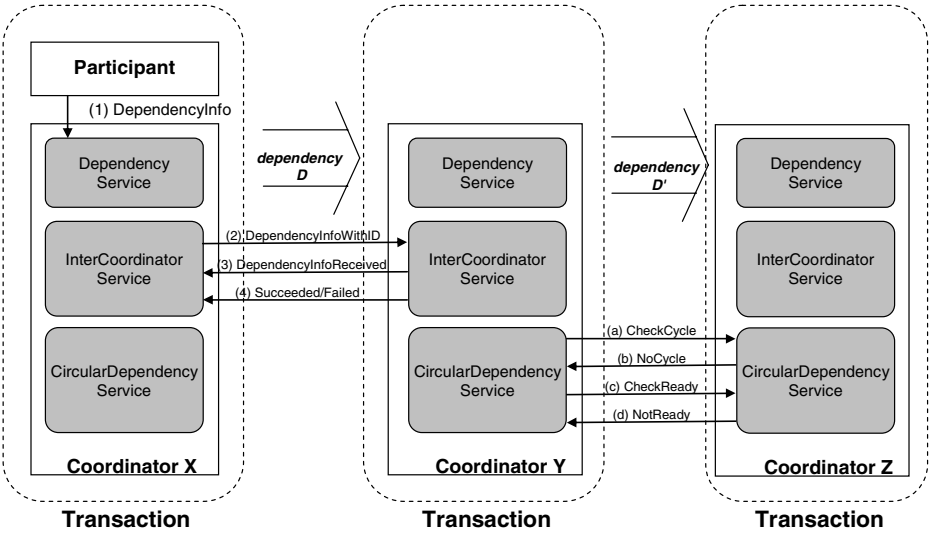

Fig. 6. The message flow of WTDP. There are two completion dependencies – D and D'. D is between transactions managed by coordinators X and Y. D' is between transactions of coordinators Y and Z.

Figure 6 shows WTDP messages and their flows. When a completion 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

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

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 completion status of the dominant transaction (4). The dependent coordinator can handle its transaction appropriately according to the completion status of the dominant transaction.

Meanwhile, the dominant transaction may additionally have a completion 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). The dominant coordinator of *D*' returns NoCycle if no cycle is found (b). Note that if a cycle exists, the CheckCycle message will finally be returned to the sender. If circular waiting is detected, the dominant coordinator sends a CheckReady message to its dominant coordinator to check whether all involved transactions are ready to be completed or not (c). Coordinators in the cycle return a NotReady message if their transactions are not ready to completed yet (d). This ready-check process is performed periodically. Detailed description of the messages and their flows are described in subsequent subsections.

5.2 Dependency Service

A participant detects completion dependencies by observing activities of operations running on itself. When a participant detects a dependent operation (which reads a value updated by other operation), it sends a DependencyInfo message to the Dependency service of the coordinator managing the transaction of the dependent operation. A DependencyInfo message contains dependency information consisting of IDs of a dominant transaction and a dependent transaction in a completion dependency relation, and the address of an InterCoordinator service in a coordinator managing the dominant transaction.

5.3 InterCoordinator Service

Four kinds of messages, DependencyInfoWithID, DependencyInfoReceived, Succeeded, and Failed, are transferred between InterCoordinator services of dominant and dependent coordinators. After receiving a DependencyInfo message from a participant, a dependent coordinator sends a DependencyInfoWithID message to its dominant coordinator. The DependencyInfoWithID message contains a dependency ID as well as dependency information included in the DependencyInfo message. The dominant coordinator replies to the dependent coordinator with a DependencyInfoReceived message, which is an acknowledgement that the DependencyInfoWithID message was received. The DependencyInfoReceived message also includes the address of the CircularDependency service of the dominant coordinator.

Succeeded and Failed messages are used by a dominant coordinator to inform its dependent coordinator of the completion status. If the dependent coordinator receives a Succeeded message, it can complete 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.

5.4 CircularDependency 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 4.4, the cycle means that circular waiting occurs. Thus, the dominant coordinator should detect such a *circular dependency*. The easiest way to detect circular dependencies would be maintaining information on all dependencies at a central single node. We can easily detect circular dependencies from consolidated information stored in the same memory space. However, this approach causes security problems. Completion 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 circular dependencies from completion dependency information distributed over multiple coordinators. The basic idea of this scheme is that we can recognize a circular dependency when a coordinator receives the token that was initiated by itself.

A CircularDependency service includes two messages, `CheckCycle` and `NoCycle`, to execute the token-based scheme. A `CheckCycle` message is a token which is traversed along coordinators. When a dominant coordinator receives a `DependencyInfoWithID` message, it initiates a new token with a unique ID. 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.

If a dominant coordinator's transaction is not dependent on other transactions, the token traversal is stopped and the dominant coordinator responds with a `NoCycle` message to its dependent coordinator. The `NoCycle` is recursively sent to the dependent coordinator and eventually is arrived at the initiating coordinator. Only after receiving every `NoCycle` message from each dominant transaction, an initiating coordinator can guarantee that there are no circular dependencies.

Once a circular dependency is detected, we need to check whether all transactions in the cycle are ready to be completed as mentioned in Section 4.4. This checking is done in a very similar way as detecting circular dependencies. A coordinator which wants to complete its transaction initiates a CheckReady message. The initiating coordinator sends the CheckReady message to every dominant coordinator. The receiving coordinator forwards the CheckReady message to its every dominant coordinator if its transaction is ready to be completed. If the CheckReady message is returned to the initiating coordinator, it is guaranteed that all transactions in the cycle are ready to be completed. Thus, the initiating coordinator can complete its transaction and notifies its dependent coordinator of the successful completion.

If the receiving coordinator's transaction is not ready to be completed, it responds with a `NotReady` message to its dependent coordinator. The `NotReady` is recursively sent to the dependent coordinator and eventually is arrived at the initiating coordinator. Now, the initiating coordinator knows that some transactions in the cycle are not ready to be completed. Thus, it should wait for a while and re-initiate a `CheckReady` message.

5.5 Validating WTDP

To validate WTDP, we prototyped a WTDP-based WS transaction management system and implemented the scenario shown in Section 2 on the system. The design and implementation for the prototyped system is described in detail in [16]. 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 CDDL and loaded into the Wood distributor's CDD before running the scenario. The scenario runs correctly and the completion dependency is identified and its information is forwarded to relevant coordinators once it happens. Figure 7 shows a screen shot of our monitoring utility which is a part of our WTDP prototype. It displays the current status of completion 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 completion dependency between them.

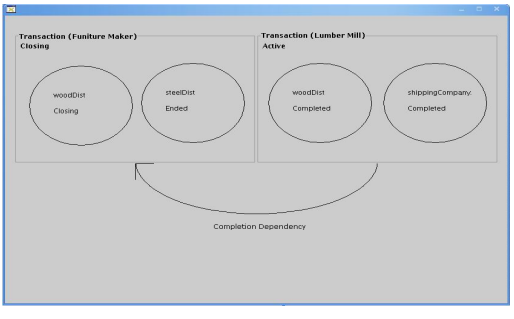

Fig. 7. A screen shot of the monitoring utility of the prototype system: two boxes mean two transactions; the furniture maker initiates the left transaction and the lumber mill initiates the right transaction. Although the right transaction has finished all its operations but it still waits for the closure of the left transaction since the operation labeled woodDist is not completed yet.

6 Conclusions

The isolation property of WS transactions is usually relaxed for a high level of concurrency. Relaxed isolation brought up the notion of completion dependency among transactions; when a transaction accesses intermediary results of another transaction, its completion depends on the completion 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 can save time and money in managing completion dependencies. Since WTDP is designed to incorporate with the WS transactions specifications, it can be easily deployed in existing WS-transactions-compliant transaction management systems.

References

1. Garcia-Molina and K. Salem. SAGAS. In Proceedings of ACM SIGMOD Conference, pages 249-259, 1987.
2. Weikum and H. J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In A. Elmagarmid (ed.): Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers, pages 515-553, 1992.
3. A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multi-database Transaction model for Interbase. In Proceedings of the 16th VLDB Conference, pages 507-518, 1990.
4. S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. A Transaction Model for Multi-database Systems. In Proceedings of the 12th International Conference On Distributed Systems, pages 56-63, June 1992.
5. A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multi-database Systems. In Proceedings of ACM SIGMOD Conference, pages 67-78, 1994.
6. IBM, Microsoft, and BEA. Web Services Transactions Specifications. http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/
7. OASIS. Business Transaction Protocol. http://www.oasis-open.org/committees/documents.php?wg_abbrev=business-transaction
8. Arjuna Technologies Ltd., Fujitsu Software, IONA Technologies PLC, Oracle Corp, and Sun Microsystems. Web Services Composite Application Framework (WS-CAF). http://developers.sun.com/techtopics/webservices/wscaf
9. R. Akkiraju, D. Flaxer, H. Chang, T. Chao, L.-J. Zhang, F. Wu, and 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, October 2001.
10. S. Tsur. Are Web Services the Next Revolution in E-commerce?. Proceedings of the 27th VLDB Conference, Roma, Italy, 2001.
11. F. Curbera, W. A. Nagy, and S. Weerawarana. Web Services: Why and How. In Proceedings of OOPLSA 2001 Workshop on Object-Oriented Web Services, Florida, USA, 2001.
12. S. Narayanan and S. A. McIlraith. Simulation, Verification and Automated Composition of Web Services. In Proceedings of WWW Conference, Honolulu, Hawaii, USA, 2002.
13. M. Pierce, C. Youn, G. Fox, S. Mock, K. Mueller, and O. Balsoy. Interoperable Web services for Computational Portals. In Proceedings of the IEEE/ACM SC2002 Conference, Baltimore, USA, November 16-22, 2002.
14. J. Eliot and B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, Cambridge, MA, 1985.
15. VMI process. http://www.vendormangedinventory.com
16. S. M. Kim, S. Choi, H. Jang, H. Kim, J. Kim, and J. Song. A Framework for Handling Dependencies among Web Services Transactions. Technical Report CS-TR-2004-207, KAIST.
17. Arjuna Technologies Ltd. ArjunaTS. http://www.arjuna.com/products/arjunats/ws.html
18. Choreology Ltd. Cohesions. http://www.choreology.com/products/index.htm
19. J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers.