CWS-TRANSACTIONS: AN APPROACH FOR COMPOSING WEB SERVICES

Juha Puustjärvi
Lappeenranta University of Technology

Keywords: Composed web services, Advanced transaction models, Semantic atomicity.

Abstract: Many transaction models have been developed for modelling composed web services. In these models subtransactions (single web services) can commit and release their resources before the whole composed transaction commits. If the whole transaction aborts, then the (semantic) atomicity is ensured by executing compensating transactions which semantically undo the effects of the committed subtransactions. However, using compensating transactions in ensuring semantic atomicity is turned out to be problematic in many cases. In order to avoid these problems we have developed a new transaction model, called CWS-transaction model, for composed web services. It deviates from other advanced transaction models in that it is not based on compensating transactions, but rather it divides the traditional *business transaction* into two successive transactions, called *request transaction* and *decision transaction.* The commitment of the request transaction ensures that the decision transaction will not fail, and so the atomicity of the CWS-transaction is ensured. In this paper we specify the components of the CWS-transaction model, their execution dependencies, the correctness criteria of the CWS-transactions and give an example of the implementation of the CWS-transaction model.

1 INTRODUCTION

A *business transaction* is an interaction in the real world, usually between an enterprise and a person or between enterprises, where something is exchanged. For example, making a room reservation on a hotel and booking a flight are business transactions.

Web services (Newcomer. 2002) provide a way for executing business transactions in the Internet. They are self-describing modular applications that can be published, located and invoked across the Web. Once a service is deployed, other applications can invoke the deployed service. In general, a web service can be anything from a simple request to complicated business process.

Another nice feature of web services is that new and more complex web services can be composed of other web services. However, in many cases composed web services are useful only if they can be processed atomically. For example, assume that a composed web service is composed of flight reservation web service and hotel web service. Now the success of the hotel reservation may be useless if the flight reservation failed.

Many transaction and workflow models have been developed for modelling the execution of composed web services, e.g., XLANG (XLANG, 2001), XAML (XAML, 2003), BTP (Business Transaction Protocol) (BTP, 2002), WSFL (WSFL, 2003) and BPEL4WS (MPEL, 2004). The cornerstone of these models is the notion of *compensation*. This means that each subtransactions (the execution of a single web service) can commit and release its resources before the whole composed transaction (composed web service) commits. If the whole transaction aborts (i.e., at least one subtransaction failed) then the (semantic) atomicity (Lynch, 1983) of the composed web service is ensured by executing compensating transactions which semantically undo the effects of the committed subtransactions.

However, using compensating transactions (Garcia-Molina, 1983) in ensuring atomicity may be problematic. To illustrate this let us consider the composed business transaction comprising of hotel room reservation and flight reservation. Now assume that the hotel reservation was successfully processed whereas the flight booking failed. So, the hotel reservation has to be rolled back by a compensating

Puustjärvi J. (2006).
CWS-TRANSACTIONS: AN APPROACH FOR COMPOSING WEB SERVICES.
In *Proceedings of WEBIST 2006 - Second International Conference on Web Information Systems and Technologies - Internet Technology / Web Interface and Applications*, pages 69-74
DOI: 10.5220/0001239400690074

transaction. Now we may encounter the following problems:

First, rolling back a business transaction is not always free of charge, and so the cancellation of the hotel reservation may give rise for a special charge.

Second, there are two semantics for the hotel reservation transaction: From composed transactions point of view the reservation is a reservation of resource which will be realized only if all the subtransaction succeed (with traditional transaction processing such a function is carried out by locks). From hotel point of view the reservation of a composed business transaction is like any reservation.

In addition, compensation is not always possible. For example, withdrawing 100 euros from account A is the compensation of deposing 100 euros on account A. However, the withdrawing will fail if the balance of account A is less than 100 euros.

In order to avoid these problems we have developed a new transaction model, called *Composed Web Service Transaction model,* or *CWS-transaction model* for short. It deviates from other advanced transaction models in that it is not based on compensating transactions, but rather it divides the traditional *business transaction* into two successive transactions, called *request transaction* and *decision transaction.* The commitment of the request transaction ensures that the decision transaction will not fail, and so the atomicity of the CWS-transaction can be ensured.

The rest of the paper is organized as follows. First, in Section 2, we specify the syntax and semantics of the CWS-transaction model. Then, in Section 3, we specify the coordination requirements for the execution of CWS-transactions. In particular, the message interchange between web services is illustrated. In Section 4, we illustrate how the request transaction and the decision transaction can be implemented in a local application. The required transactions are presented by an SQL-like notation. Section 5 concludes the paper by discussing the advantages and limitations of the CWS-transaction model.

2 THE STRUCTURE OF THE CWS-TRANSACTIONS

In this section we specify the components of the CWS-transaction model and their execution dependencies.

In our terminology we refer by the term *web service transaction*, or WS-transaction for short, to the execution of a web service. So, for example, WS-transaction is an execution of a web service which makes a reservation on a hotel through. Further WS-transaction comprises of one or more *web service operations*, or *WS-operations* for short. For example, requesting the prises of hotel rooms and making the actual room reservation are the WS-operations comprising a WS-transaction.

Further, we make the difference between *read operations* and *update operations* of a WS-transaction, e.g., requesting the prises is a read operation whereas making the actual reservation is an update operation.

Transactional feature (Gray and Reuter, 1993) in the context WS-transactions means that all or none of its update operations are executed. For example, if the function of a flight reservation WS-transaction is to make a reservation on flight A and B, then both or none of the reservations are done.

Even though the WS-transaction model is useful for executing and analyzing single web services, it is not enough powerful for modelling composed web services. Therefore we use WS-transaction as components in the CWS-transaction model.

The structure of the CWS-transaction is presented in Figure 1.

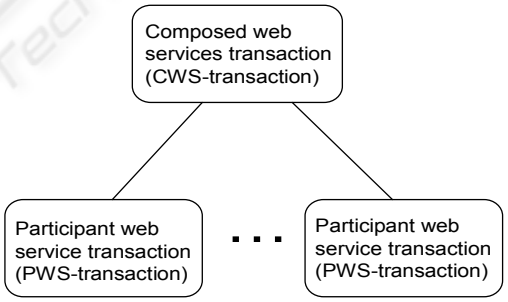

Figure 1: The structure of the CWS-transaction.

The function of the root (CWS-transaction) is to coordinate the execution of the leaf transactions (Participant Web Service transactions, or PWS-transactions for short).

Further each PWS-transaction is divided into a *request transaction* and a *decision transaction* (Figure 2). The function of a successfully executed request transaction is to ensure that the decision transaction will not semantically fail. The decision transaction will only be executed if the corresponding request transaction is successfully executed. Further, each decision transaction either

confirms or cancels the request. Note that technically request and decision transactions are normal WS-transactions.

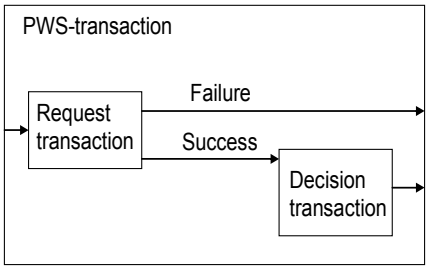

Figure 2: The execution dependencies of the PWS-transaction.

The execution dependencies of the PWS-transaction are presented in Figure 3.

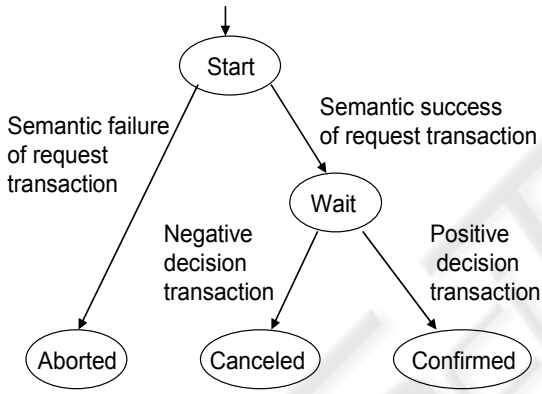

Figure 3: The states of a PWS-transaction.

In order to illustrate the execution of the CWS-transactions let us assume that a business_trip_reservation is a CWS-transaction (Figure 4). Its PWS-transactions are hotel_reservation_transaction and flight_reservation transaction, which in turn are divided into request transaction and decision transaction.

After the execution the business trip transaction it is either in aborted state or committed state (Figure 5). It is in the aborted state, if one or more request transaction failed. This may happen for example when the flight or the requested hotel is fully booked.

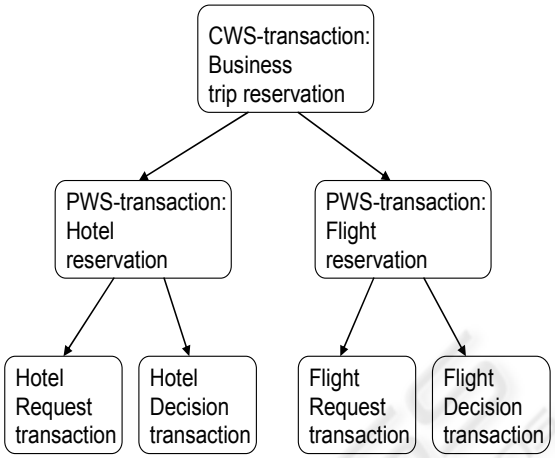

Figure 4: The structure of a CWS-transaction.

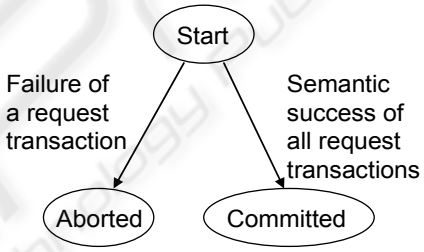

Figure 5: The states of a CWS-transaction.

3 SUPPORTING CWS-TRANSACTIONS

Based on the concepts presented in Section 2 we can now specify the correctness criteria of the CWS-transactions. They are the followings:

C1. Each CWS-transaction either commits or aborts.

C2. If the request transaction of each PWS-transaction is successfully executed, then their positive decision transaction is also executed (i.e., the CWS-transaction is committed).

C3. If the CWS-transaction aborted, then each of its PWS-transaction either aborted or cancelled.

Enforcing these constraints requires the coordination between the CWS-transaction and its PWS-transactions. In the following protocol description to illustrate the functions of the components, we call

the root as coordinating web service while other web services we call participating web service. This kind of communication architecture can be presented as graph where nodes are Web services (Web services, 2002) and directed edges represent SOAP-messages (SOAP, 2002) (Figure 6).

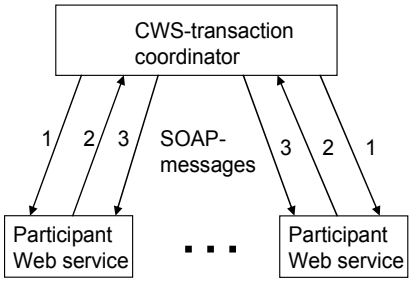

Figure 6: The communication structure of CWS-transaction.

For simplicity we first present the atomicity protocol assuming that there are no communication failures. In such a case the atomicity protocol of composed web services goes as follows:

1. The CWS-transaction coordinator sends the *request message* to participant web services.

2. A participant web services execute the request transaction. If the execution failed the participant web service sends to CWS-transaction coordinator the *failure message*; otherwise it sends the *success message.*

3. If the CWS-transaction coordinator has received the *success message* from all participant web services, then it sends the *positive decision message* to each participant web service (i.e., requests to execute the positive decision transaction). Otherwise it sends the *negative decision message.*

Note that, this protocol will terminate only if all messages are received. There are two places where a service is waiting for a message: in the beginning of steps 2 and 3. In the beginning of step 2, a participant web service waits for a request from the CWS-transaction coordinator. In step 3, the CWS-transaction coordinator is waiting for the decision (positive or negative) from all the participant web services.

We say that when a service must await the repair of failures before proceeding, the service is blocked. Blocking is undesirable, since it can cause services to wait for an arbitrarily long period of time. In order that a blocked service can proceed it must communicate with the CWS-transaction coordinator. This kind of communication is carried out in a *termination protocol.* Participant web service activates a termination protocol when it has been waiting a predetermined time for a message. Our termination protocol of the atomicity protocol goes as follows:

1. The participant web service sends *decision-request-message* to the CWS-transaction coordinator.

2. The CWS-transaction coordinator sends the *response-message* to the participant web service.

The participant web service repeats the request if it has not received the response in a predetermined time period. The CWS-transaction coordinator is always able to response to the request as it has no uncertainty period. Uncertainty period is the time period between the moment a participant web service sent the success message to the CWS-transaction coordinator and the moment it has received the decision message. During the uncertainty period the participant web service does not know whether the CWS-transaction coordinator will eventually commit or abort the CWS-transaction.

4 IMPLEMENTING PWS-TRANSACTIONS

A salient feature of the CWS-transaction model is that if a participant web service sends the *success message* to the CWS-transaction coordinator, then it is committed to execute the decision (either positive or negative) transaction. The problem here is how to ensure that the decision transaction will not semantically fail in executing the positive decision transaction. We illustrate our used technology by a web service of an imaginary airline (Figure 7). Web services are described in WSDL (WSDL, 2001) but here we omit the descriptions.

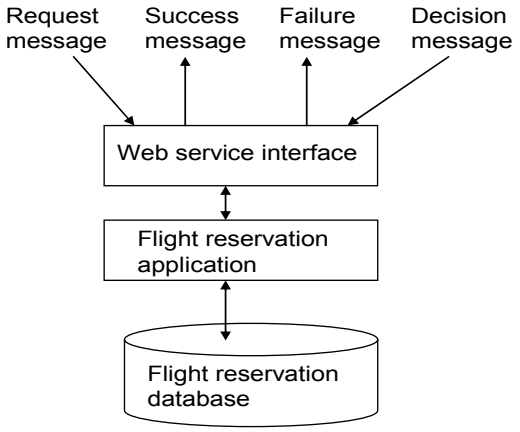

Figure 7: Web service of a reservation system.

Assume that the *Flight reservation database* is comprised of the three relations: *Reservations* (Figure 8), *PreliminaryReservations* (Figure 9) and *ReservationStates* (Figure 10).

Relation *Reservations* includes reservations. Relation *PreliminaryReservations* includes the reservations inserted by the transaction triggered by the Request-message. They are called *preliminary reservations* as they will be changed to reservations by the positive decision transaction or they will be deleted by the negative decision transaction. That is, the positive decision transaction deletes the preliminary reservation and makes a reservation while the negative decision transaction only deletes the preliminary transaction.

Reservations	customer	flight	resSeats
	Smith	A -123	1
	Jones	C -345	3
	Taylor	B -234	1
	Cooper	A -123	2

Figure 8: Relation Reservations.

PreliminaryReservations	customer	flight	preResSeats
	Robinson	A -123	1
	Harrison	C -345	2

Figure 9: Relation PreliminaryReservations.

ReservationsStates	flight	resNum	preResNum	maxRes
	A -123	3	1	60
	B -234	1	0	50
	C -345	3	2	90

Figure 10: Relation ReservationStates.

Relation *ReservationStates* captures the information of preliminary reservations (attribute *preResNum*) and reservations (attribute *resNum*). Attribute *maxRes* indicates the number of reservations that can be made on flights. In order to avoid overbookings the specification of the relation *ReservatioState* includes the following consistency constraint:

```
CONSTRAINT BookingWatch
CHECK (resNum + preResNum <= maxRes)
```

Note that this constraint also ensures that each preliminary reservation can be changed to reservation. That is, changing a preliminary reservation into reservation cannot fail as a result of unreserved seats.

The specifications of the transaction making the preliminary reservation and the positive and negative decision transactions are given below. The *Request message* (Figure 7) activates the *PreReservation transaction,* and the *Decision message* (Figure 7) activates either the *PositiveDecision transaction* or the NegativeDecision transaction. Here we use an SQL-like notation which deviates from SQL in that we omit certain features, which are irrelevant from the illustrative point of view, e.g., we omit the check of the SQLSTATE-variable.

```
PreReservation (customerID, flightId,
resSeats)
Begin transaction
    INSERT INTO PreliminaryReservations
        VALUES (:customerID, :flightId,
:reseats);
    UPDATE ReservationStates
        SET preResNum = preResNum +
:reseats
        WHERE flight = :flightId;
End transaction

PositiveDecision (customerID, flightId,
resSeats)
Begin Transaction
    UPDATE ReservationStates
```

```
        SET  preResNum  =  preResNum  -
:reseats,
        resNum = resNum + :resSeats

        WHERE flight = :flightId;
    INSERT INTO Reservations
        VALUES (:customerID, :flightID,
:reseats):
    DELETE FROM PreliminaryReservations
        WHERE fligh = :flightId AND
                    customer          =
:customerID;
End transaction

NegativeDecision (customerID, flightId,
resSeats)
Begin Transaction
    UPDATE ReservationStates
        SET  preResNum  =  preResNum  -
:reseats,
        WHERE flight = :flightId;
    DELETE FROM PreliminaryReservations
        WHERE fligh = :flightId AND
                    customer          =
:customerID;
End transaction
```

5 CONCLUSIONS

A goal of web services is to achieve universal interoperability between applications by using web standards. However, the full potential of universal interoperability will be achieved only when web services can be integrated in a transactional way. In order to achieve this goal many *transaction models* have been developed. In these models compensating transactions are used to ensure the semantic atomicity of transactions spanning over many sites.

In many cases the use of compensating transactions has turned to be problematic. In order to avoid these problems we have developed the CWS-transaction model, which does not use compensating transactions. In contrast it divides the traditional business transaction into two successive transactions, called request transaction and decision transaction. The commitment of the request transaction ensures that the decision transaction will not fail, and so the semantic atomicity of the CWS-transaction is ensured. In addition the termination protocol ensures that CWS-transactions tolerate also communication and site failures.

A restriction of our approach is that the support of CWS-transactions requires minor modifications on local applications. In particular, the updates of the request transaction must be stored in a stable storage, typically on the database. This in turn requires creating new data structures, e.g., relations.

REFERENCES

Newcomer E., 2002. Understanding Web Services Addison-Wesley.

XLANG, 2001. XLANG–Web Services for Business Process Design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.

XAML, 2003. Transaction Author Markup Language (XAML). http://xml.coverpages.org/xaml.html.

BTP, 2002. BTP- Business Transaction Protocol, http://www.oasis-open.org/committees/business-transactions/documents/primer/.

WSFL, 2003. WSFL- Web Services Flow Language. http://www.ebpml.org/wsfl.htm

BPEL, 2004. BPEL4WS – Business Process Language for Web Sevices. http://www.w.ibm.com/developersworks/webservices/library/ws-bpel/.

Lynch N., 1983. Multilevel atomicity – a new correctness criterion for database concurrency control. ACM Transactions on Database Systems, 8(4):65-76.

Garcia-Molina H., 1983. Using semantic knowledge for transaction processing in a distributed database. ACM Transactions on Database Systems, 8(2):186-313.

Gray, J. & Reuter A. 1993. Trasaction Processing: Concepts and Techniques. Morgan Kaufman.

Web services, 2002. Web Services Activity. http://www.w3.org/2002/ws/.

SOAP, 2002. SOAP – Simple Object Access Protocol. http://www.w3.org/TR/SOAP/.

WSDL, 2001. WSDL- Web Services Description Language. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.