A Hierarchical Model for Transactional Web Service Composition in P2P Networks

Joyce El Haddad, Maude Manouvrier, Marta Rukoz
A Hierarchical Model for Transactional Web Service Composition in P2P Networks

Joyce El Haddad1 Maude Manouvrier1 Marta Ruko2

1LAMSADe, University of Paris Dauphine, France
{elhaddad,manouvrier}@lamsade.dauphine.fr

2University of Paris X Nanterre, France
marta.castillo@u-paris10.fr

Abstract

The recent approaches for Web services composition tend to integrate heterogeneous business processes executed in Peer-to-Peer networks. In such networks, component Web services are invoked on independent peers and are orchestrated according to the transactional requirements defined by the designers or the users of the composite Web service. Since component Web services can be dynamically invoked and are generally implemented as black boxes, concurrency between them may appear. This paper presents the transactional execution model of composite Web services exploiting the transactional properties of their component Web services. The proposed concurrency control is ensured by a decentralized serialization graph based on an optimistic protocol and on the hierarchical structure of the composition. The globally correct execution of the composite Web service is achieved by communication among dependent subtransactions and the peers they have accessed.

1. Introduction

With the proliferation of e-business technologies, service-oriented computing is becoming increasingly popular. Access to data and documents is provided by services which can range from simple read/write operations on data items to complex business functions like scheduling a trip. An important challenge is to combine service invocations into a coherent whole by means of composition. Services that enter into compositions with other services may have transactional properties. These transactional properties may be exploited in order to derive composite services which themselves exhibit certain transactional properties.

This poses, however, several new research problems. For instance, how can one at the same time create Web service compositions tailored to each user and ensure their correct execution. Moreover, this kind of “à la carte” composition brings about issues such as managing concurrent access to resources and ensuring a correct execution in accordance with users’ requests and preferences. These issues are even more difficult to resolve in a peer-to-peer environment that inherently lacks global control.

Most approaches to Web service composition can be classified into two main classes: those based on workflows [1] [2] [8] and those based on advanced transactional models [6] [7] [14] [16]. The first class enables a certain degree of flexibility, but lacks sound mechanisms for correctness and concurrency control. On the other hand, advanced transactional models [9] [11] handle concurrency but are found lacking functionality and performance when used for applications that involve dynamic composition of heterogeneous services in a peer-to-peer context. Their limitations come mainly from their inflexibility to incorporate different transactional semantics as well as different interactions patterns into the same structured transaction.

To the best of our knowledge, defining a transaction with a particular set of properties and ensuring that every execution will preserve these properties remains a difficult and open problem. Our work is a step towards solving this problem. As will be seen, we support user-tailored composition, by creating a multi-level hierarchy of Web services, similar to a workflow specification, where composition can be specified using logical connectors. At the same time, we take advantage of results in database open nested transaction protocols [15] to manage concurrent accesses and ensure correctness. In this paper, we propose a transactional approach for reliable Web services compositions by ensuring the constraints required by the users. From a transactional point of view, we consider a composite Web service as a structured transaction and component Web services as
subtransactions. We use the user's requirements as a correctness criterion. Indeed, the constraints and preferences specified by the users over the set of services that participate in a transaction have to be a part of the execution of the composite service.

The remainder of the paper is organized as follows. Section 2 introduces a motivating example and gives the main point which has driven our approach. In section 3, we present the composite Web service transaction model. Section 4 describes our concurrency control of the composite Web services model and illustrates how our approach proceeds, using the user's requirements, to compose reliable Web services. In section 7, we discuss some related work. Section 8 concludes and gives perspectives.

2. Motivating example

In order to analyze the requirements of composite Web services transactions, we consider as a working and a motivating example a scenario of an online trip reservation. It involves three Web-based autonomous businesses that provide specialized services: \( S_1 \) providing flight reservation, \( S_2 \) providing hotel reservation and \( S_3 \) providing B&B accommodation. These services can be composed into several composite Web services in order to provide packages which can be chosen by the customers such as:

- **Package1** that consists of booking a flight and a reservation of a room hotel;
- **Package2** that consists of booking a flight and a reservation of either a room in an hotel or in a B&B;
- **Package3** that simply consists of booking a flight ticket.

As illustrated in Figure 1, **Package1** is a workflow skeleton composed of Web services \( S_1 \) and \( S_2 \). **Package2** a workflow skeleton composed of Web services \( S_1 \), \( S_2 \) and \( S_3 \). **Package3** is the elementary Web service \( S_3 \).

**Example.** Let us consider the example of a trip scheduling where a customer wants to reserve for three different persons at the same time. The trip scheduling consists of booking a **Package1** for a first person, a **Package2** for a second person and only a flight for the customer himself. In this example, the customer wants to be sure that the trip is valid if he has his flight ticket and at least one of the two other persons could have his package. As depicted in Figure 2, the reservation request of the customer (**Package1** OR **Package2**) AND **Package3** is defined using the AND-split and the OR-split patterns. If the first part of the trip scheduling fails (i.e. **Package1** OR **Package2** is false) then the customer is stuck with a flight ticket. In that case, nobody travels and it is a valid option to cancel the whole trip. However, if someone of the first part has its package valid and the customer has his flight ticket too, then the whole trip is valid.

![Figure 1 - Packages for online trip reservations.](image1)

![Figure 2 - Online trip reservation request.](image2)
services with traditional ACID propriety (i.e. atomicity is all-or-none) is:

- the three flight reservations, the two hotel reservations or one hotel reservation and one B&B reservation are committed.

However, the customer wishes to establish a transactional dependency between the services so that the following defined outcomes of the combined use of the Web services are possible:

- the three flight reservations, the two hotel reservations or one hotel reservation and one B&B reservation are committed;
- the two flight reservations and either the one hotel reservation or the one B&B reservation are committed;
- the two flight reservations and the one hotel reservation are committed.

This example motivates our research to provide users with a mean to express their acceptable atomicity level and the correct execution of the composite service regards to their preferences. In the rest of the paper we present an appropriate transactional model for the composition of Web services with transactional properties that solves the above issues.

3. Hierarchical transactions for Web services

This section summarizes the basic assumptions of our approach.

3.1. Architecture

We adopt peer-to-peer architecture as it is widely used for various Web-based distributed applications. Peer-to-peer has many advantages including, dynamic communication, enhanced reliability without suffering from single point of failures, and load sharing among peer systems. The proposed architecture comprises various components including user application (i.e. the customer), service packages (i.e. composite Web services) and system model (i.e. peer or component Web services). In the following, we describe these components.

3.1.1. User application and service packages. A user application component acts as a consumer of the (composite) Web service. It invokes operations on the (composite) service in order to perform various tasks. For example, a user may use the (composite) service to book a flight, reserve a room in a hotel or in a B&B. In other terms, each composite Web service can be viewed as a black box. Its interface provides several packages, which can be selected and/or combined by the users. A service package corresponds to the invocation of one or several component Web services.

Component Web services of a composite service are generally developed by individual organizations. In the proposed model each (composite) service is self-coordinated, autonomous and performs various functions including: (i) receiving service activation requests from the user application or from a peer, (ii) invoking operations on the underlying component services, (iii) receiving notification of the execution completion from peer and changing its state accordingly, (iv) exchanging input and output data and (v) other message communication such as reporting failures, cancellation of requests, and stopping the service execution.

3.1.2. System Model. Our model is based on the peer-to-peer environment of [13] with the following assumptions:

- Without loss of generality, each peer provides one service. This service can be invoked using its interface and is executed as local database transaction.
- The peers are independent and services are not replicated on different peers.
- A peer may provide a compensation service that semantically undoes the effect of the invocation of the original service. The operation of the compensation service strongly depends on the semantics of the original service. The compensation might also be an "empty" service.
- Each service provides one of two kinds of resources: resources handling acquisition operations (e.g. flight tickets) and resources handling read/write operations (e.g. bank account). For the latter, each service maintains only one resource. For the former, each service can maintain several resources. In both cases and for each resource, a peer maintains a log file where it stores the identifiers of the transactions invoking the service on this resource. Using this information, a peer can derive conflicts between transactions that have invoked the same service on the same resource.

3.2. Transactional composite Web service

Returning to our online trip reservation scenario from above, where a customer wants to schedule a travel according to its following restriction: (Package1 OR Package2) AND Package3. Usually, interactions between the services that offer the resources and the users interested in them are encapsulated within a transaction. Since our system model is based on a peer-to-peer network, the open nested transaction model [15] seems to be best suited. Under this model, a transaction can launch any number of subtransactions, which, in turn, can launch any number of subtransactions, thus forming a
transaction hierarchy. Each subtransaction can in turn be an open nested transaction. The transaction which is not enclosed in any transaction is called the root transaction. Each leaf subtransaction is viewed as a normal flat transaction in the system and corresponds to a service invocation. That is, only leaves of the transaction hierarchy can perform the Web services invocations and are executed independently. Non-leaf subtransactions organize the control flow and determine when to execute subtransactions. Transactions having subtransactions are called parents, and their subtransactions are their children. In the following, we use the term transaction to denote both root transaction and subtransactions. The transaction identifiers reflect the hierarchy: the children of a transaction $T_j$ are identified by $T_{jk}$, $k \geq 1$.

**Example.** Consider again the example of the online trip reservation. As depicted in Figure 3, the root transaction $T_j$ corresponds to the customer request and the logical expression $(T_{1j} \lor T_{12j} \land T_{13j})$ to its restriction. This transaction is a hierarchy of three subtransactions: $T_{1j}$ modelling Package1, $T_{12j}$ modelling Package2 and $T_{13j}$ modelling Package3. The logical expressions of $T_{15}$, $T_{12j}$, $T_{13j}$ correspond to the packages' workflow patterns.

![Figure 3- Example of a transaction hierarchy.](image)

To exploit the inherent potential of open nested transactions and their advantages, the degree of intra-transaction parallelism should be as high as possible. In such transactional model only siblings may be performed concurrently. Since some transactions in a hierarchy are executed in parallel, concurrency control among them is needed.

### 3.3. Composite Web service orchestration

Web service composition is generally accomplished in different phases. The first phase handles the orchestration of the component Web services which are discovered. It selects appropriate services and constructs the execution flow for those services. The second phase presented in the next Section concerns the concurrency control of the composite Web service execution.

Component Web services are executed according to the execution order required by the composite service. For instance, in the online travel reservation example, a flight reservation service could be executed prior to the execution of a hotel reservation service. The execution flow of a composite service is constructed such that it conforms to the execution requirements of the customer (i.e user application). We define some basic rules in order to construct an execution flow for the component services and the related transactions:

- **Sequential ($S_2/S_3$):** A component service, $S_2$, must follow the execution of another component service, $S_3$. For instance, a flight reservation service should be executed before a B&B accommodation service which should be executed after the execution of an hotel reservation service. Therefore, as depicted in Figure 3, transaction $T_{12j}$ corresponding to the hotel reservation service is executed in sequential with $T_{123j}$ that corresponds to the B&B accommodation service.

- **Parallel ($S_2/S_3$):** This allows component services, $S_2$ and $S_3$ to be executed concurrently. For example, the flight reservation service and the hotel reservation service can be executed in parallel. Therefore, as depicted in Figure 3, transaction $T_{11j}$ corresponding to the flight reservation service is executed in parallel with $T_{112j}$ that corresponds to the hotel reservation service.

### 4. Execution model

This section addresses the concurrency control issue between the transactions of the hierarchy. This is ensured by the collaboration of peers and of transactions. In fact, we assume that there is a conflict between two or more transactions when they invoke the same service on the same resource. Our approach is based on an optimistic concurrency control. Thus, a transaction invokes a service without checking for conflicts, i.e. conflicts are detected afterwards. Indeed, each transaction of the hierarchy manages its own serialization graph.
comprising the conflicts in which the transaction is involved in.

4.1. Peer resource management

As mentioned above, for each resource of a service $S$, a peer $P_i$ maintains a log file where it stores the identifiers of the active transactions invoking $S_i$ for this resource. Since service $S_i$ maintains either $n_i$ acquisition resources (i.e. $n_i=\overline{0}$) or one read/write resource (i.e. $n_i=\overline{1}$), an instance of a log file for a resource contains the queue of transaction invocations.

When the peer service provides $n_i=\overline{0}$ resource acquisitions, the completion of a transaction decreases the value of $n_i$. The completion of $n_i$ transactions involves the failure of all the transactions following the completed ones on the same service invocation.

When the peer service provides a read/write resource (i.e $n_i=\overline{1}$), the failure of a transaction $T_i$ decreases the value of $n_i$, and involves the failure of all the following transactions invoking the same service.

4.2. Peer transactional layer

The message exchange between peers and leaf transactions performs the following:

- When service $S_i$ of peer $P_i$ is invoked by an active transaction $T_j$, then if the execution of $S_i$ is not possible ($n_i=\overline{0}$), peer $P_i$ sends to $T_j$ an error message. Otherwise, peer $P_i$ stores the service invocation of $T_j$ into its local log file, executes $S_i$, and sends its log file to $T_j$ with the service result. Figure 3 presents an example. The log file of service $S_i$ points out that transaction $T_{111}$ precedes transaction $T_{113}$ which precedes transaction $T_{112}$ on the invocation of service $S_i$. When peer $P_i$ receives the invocation of service $S_i$ by transaction $T_{121}$, it sends to $T_{121}$ the result service invocation and its log file indicating that transactions $T_{111}$ and $T_{113}$ have invoked the same service on the same resource before $T_{121}$.

- When peer $P_i$ receives a message indicating that a transaction $T_j$ completes (resp. fails), $P_i$ updates its log file. Then, peer $P_i$ analyzes its log file and informs all the transactions following $T_j$ on the same service resource that either they have to fail, if $n_i=\overline{0}$, or that $T_j$ is completed (resp. has failed), if $n_i>\overline{0}$. Moreover, if a compensate service exists, the failure of $T_j$ involves the invocation of the compensate service by a new transaction. For example, let suppose peer $P_i$ of Figure 3 receives a message indicating that transaction $T_{111}$ completes. If only one flight seat is available ($n_i=\overline{1}$ before $T_{111}$ completes), then peer $P_i$ decreases the value of $n_i$ and sends to $T_{113}$ and $T_{121}$ a message indicating they have to fail. In return, if $P_j$ receives a message indicating that transaction $T_{113}$ is failed, then it invokes a compensation service, if such service exists, and informs $T_{111}$ and $T_{121}$ that $T_{113}$ has been failed.

4.3. Serialization graph construction

As we mentioned before, we use an optimistic concurrency control, where each transaction manages its own serialization graph. This graph allows a transaction to detect non-serializable execution between their subtransactions. In this case, one of its subtransactions has to fail.

The graph of a leaf transaction is built from the log file sent by the peers whose services have been invoked. The serialization graph of a non-leaf transaction is induced by the graphs sent by its children. In both cases, two steps appear. Firstly, a transaction $T_j$ updates its graph by merging its previous graph with the graph(s) of the log file received. Secondly, the transaction $T_j$ replaces the identifiers of the transactions appearing in the graph by the identifiers of transactions appearing on the same level than $T_j$ in the hierarchy (i.e. reducing the size of transaction identifiers from the end). The serialization graphs of the example transactions of Figure 3 are represented in Figure 4. The graph of leaf transaction $T_{11}$ is computed from the message of peer $P_j$ indicating that the invocation of $S_j$ by $T_{11}$ is preceded by the invocation of the same service by $T_{111}$ on the same resource. Thus, the graph of $T_{11}$ is $T_{111} \rightarrow T_{11}$. After restructuring the transaction identifiers, the graph becomes $T_{11} \rightarrow T_{11}$.

The inferred serialization graphs are transferred bottom-up from the peers to the leaves and then up to the root via the intermediate transactions.
Figure 4. Serialization graphs of the transactions.

When a transaction detects a cycle in its serialization graph, two cases appear. First, the logical expression of the transaction is a conjunction of its subtransactions. In this case, any failure of one of its children involves a failure of the transaction. In the second case, if a disjunction appears in the logical expression of the transaction, then a victim can be chosen among the children of the transaction. The chosen victim transaction must fail in order to delete the cycle in the graph, after receiving an “Error!” message from its parent transaction.

4.4. State transitions of transactions

The state diagram of a Web service transaction is represented by Figure 5. It contains seven states. The Initial state of the diagram means that the transaction does not have started execution yet.

When a transaction changes its state, it informs its parent about this change by sending its serialization graph. In addition, leaf transactions inform the peers whose services have been invoked.

![State diagram of a WS transaction](image)

Figure 5. State diagram of a WS transaction.

In certain cases, a transaction changes its state if and only if the following two conditions are valid: (1) the evaluation of the logical expression of the transaction is true and (2) no cycle appears in the serialization graph of the transaction. In the following, we call this verification Execution Control Consistency (ECC). The logical expression of a leaf transaction is true if the service has been correctly invoked by the transaction. The transitions between states depend on the messages received or sent by a transaction and on the position of the transaction in the hierarchy, as explained below.

Transition between Initial state and Active state:

- For the root transaction: When the user’s request is initiated the root transaction becomes Active, sends an “Execute!” message to all its children and waits for their answers.
- For leaf transactions: After receiving an “Execute!” message from its parent, a leaf transaction becomes Active, sends an invocation message to the peer and waits for the peer’s answer.
- For non-leaf transactions: After receiving an “Execute!” message from its parent, a non-leaf transaction broadcasts the “Execute!” message to its children and waits for their answers.

Transition between Active state and Executed state:

- For leaf transactions: After receiving a message from the peer indicating that its service has been correctly executed. The transaction updates its serialization graph from the informations sent by the peer, sends it to its parent with an “Executed” message and changes to Executed state.
- For non-leaf transactions: After receiving a message “Executed” from (at least one of) its children and if the ECC is verified. Each non-leaf transaction (except the root), sends an “Executed” message to its parent and changes to Executed state. For example, if transaction $T_{12}$ receives two messages indicating that both sub-transactions $T_{121}$ and $T_{122}$ (or $T_{121}$ and $T_{123}$) have been executed then $T_{12}$ changes to Executed state.

In the rest, each time that a transaction evaluates its logical expression, it sends a message to each of its children transactions. This message is either an “Error!” message intended to the children of the invalid part of its logical expression, or a “Complete!” or “Commit!” message intended to the children of the valid part of its logical expression.

Transition between Executed state and Wait-To-Completed state
• For the root transaction: After sending a “Complete!” message to its children, the root transaction goes to \textit{Wait-to-Completed} state.

• For the other transactions: When receiving a “Complete!” message from its parent transaction. In addition, each non-leaf transaction sends a “Complete!” or an “Error!” message to its children, depending on the evaluation of its logical expression.

Transition between \textit{Wait-To-Completed} state to \textit{Completed} state:
• For all the transactions: The transition to state \textit{Completed} is possible if no other transaction precedes the transaction in its serialization graph. In the following, this situation is called No Dependence.

• For leaf transactions: After sending a message “Completed” to the peer whose service has been invoked, and if there is No Dependence. For example, transactions $T_{111}$, $T_{122}$ and $T_{123}$ can go to state \textit{Completed} after receiving a “Complete!” message from their parent transaction, because No Dependence appears.

• For non-leaf transactions: After receiving a message “Completed” from (at least one of) its children and if the ECC is verified and if there is No Dependence.

Transition between \textit{Completed} state and \textit{Commited} state:
• For the root transaction: As soon as the transaction has sent “Commit!” to its children.

• For the other transactions: After receiving a “Commit!” message from its parent transaction. This message is broadcasts to the \textit{Completed} children by non-leaf transactions or to the peers by leaf transactions.

Transition between \textit{Active/Executed/Wait-To-Completed/Completed} states and \textit{Failed} state:
• For leaf transactions: After receiving an “Error!” message from the peer or from its parent transaction. For example, if there is only one flight seat available, peer $P_i$ sends an “Error!” message to transactions $T_{111}$ and $T_{122}$ after receiving a “Completed” message from $T_{111}$. When a peer allows the compensation of transactions, the transition to state \textit{Failed} of a leaf transaction involves the execution of a compensation transaction (invoking a compensation service).

• For the other transactions: After receiving an “Error!” message from its parent, or if the evaluation of the logical expression of the transaction is false. The logical expression evaluation is done by a transaction after receiving a message from one of its children or after choosing one of its children to be a victim in case of cycle in its serialization graph. For example, when $T_{13}$ informs its parent transaction it has failed, the state of $T_3$ becomes Failed because the logical expression of $T_3$ becomes false after the failure of $T_{13}$.

5. Related work

Designing a set of service to achieve a composite Web service has been tackled by workflow systems and by advanced transactional models. These two classes of approaches are complementary but suffer from concurrency control for the first one and from inflexibility of compositions for the second one.

Workflows are flexible but lack transactional reliability. To overcome this limitation, the approach of [1] [2] proposes to validate the transactional requirements of the user once a composition of Web service has been created. On the other hand, the approach of [8] integrates the user transactional requirements as a part of the composite Web service building process.

Among advanced transactional models, emerging standards, such as WS-Transaction [4], BTP [10] [12], and WS-TMX [3], propose two-phase centralized orchestration of composite Web services [5]. To overcome the bottleneck associated with a centralized controller, several approaches [6] [13] [14] propose a decentralized orchestration of composite Web services. In [6], the authors use an extension of the two-phase coordination protocol. In addition, their approach allows the user to express maximality and minimality constraints over the set of services expected to the validation phase. However, this model is limited to a sequential execution of transactions. In contrast, in [14], the authors present a multi-level model for service composition that does not support users’ constraints.

Another related work is presented in [13] describing a decentralized coordination of web services in peer-to-peer environment. This approach is based on serialization graph testing but does not take into account user preferences and is also limited to a sequential execution of transactions.

As a consequence, none of these aforementioned approaches is able to implement an “à la carte” composite web service, such as expressed our online trip example. Indeed, our model allows users to express their constraints over the set of composite Web services. It is based on open nested transaction model [15] in a peer-to-peer context as for [13]. Moreover, our approach takes advantage from the transaction hierarchy to achieve global concurrency control.
6. Conclusion

This article has presented a transactional execution model for user-tailored composite Web services. Since component Web services can be dynamically invoked and are generally implemented as black boxes, concurrency between them may appear. To deal with this, we use an optimistic protocol and the hierarchical structure of the composition to ensure global concurrency control. The globally correct execution of the composite Web service is achieved by communication among dependent subtransactions and the peers they have accessed.

This article has presented the basic concepts of our approach. For the moment, the user’s request is translated into a logical expression over the transaction hierarchy, without any analyzing of its semantic. This issue is our future research challenge.

Acknowledgements

The authors would like to thank Claudia Bauzer-Medeiros (Univ. Campinas - Brazil) for her helpful suggestions.

References


