

Dependability in the Web Services Architecture

Ferda Tartanoglu¹, Valérie Issarny¹,
Alexander Romanovsky², and Nicole Levy³

¹ INRIA Rocquencourt

78153 Le Chesnay, France

{Galip-Ferda.Tartanoglu, Valerie.Issarny}@inria.fr

<http://www-rocq.inria.fr/arles/>

² University of Newcastle upon Tyne

School of Computing Science, NE1 7RU, UK

Alexander.Romanovsky@newcastle.ac.uk

³ Laboratoire PRISM, Université de Versailles Saint-Quentin-en-Yvelines

78035 Versailles, France

Nicole.Levy@prism.uvsq.fr

Abstract. The Web services architecture is expected to play a prominent role in developing next generation distributed systems. This chapter discusses how to build dependable systems based on the Web services architecture. More specifically, it surveys base fault tolerance mechanisms, considering both backward and forward error recovery mechanisms, and shows how they are adapted to deal with the specifics of the Web in the light of ongoing work in the area. Existing solutions, targeting the development of dependable composite Web services, may be subdivided into two categories that are respectively related to the specification of Web services composition and to the design of dedicated distributed protocols.

1 Introduction

Systems that build upon the Web services architecture are expected to become a major class of wide-area distributed systems in the near future. The Web services architecture targets the development of applications based on XML-related standards, hence easing the development of distributed systems through the dynamic integration of applications distributed over the Internet, independently of their underlying platforms.

A Web service is a software entity deployed on the Web whose public interface is described in XML. A Web service can interact with other systems by exchanging XML-based messages, using standard Internet transport protocols. The Web service's definition and location (given by a URI) can be discovered by querying common Web service registries. Web services can be implemented using any programming language and executed on heterogeneous platforms; as long as they provide the above features. This allows Web services owned by distinct entities to interoperate through message exchange.

Although the modularity and interoperability of the Web services architecture enable complex distributed systems to be easily built by assembling several component services into one composite service, there clearly is a number of research challenges

in supporting the thorough development of distributed systems based on Web services. One such challenge relates to the effective usage of Web services in developing business processes, which requires support for composing Web services in a way that guarantees dependability of the resulting composite services. This calls for developing new architectural principles of building such composed systems, in general, and for studying specialized connectors "glueing" Web services, in particular, so that the resulting composition can deal with failures occurring at the level of the individual component services.

Several properties of the Web services architecture must be taken into account while addressing the above issues. Web services are decentralized in architecture and in administration. Therefore, individual Web services can have different characteristics (e.g., transactional supports, concurrency policies, access rights), which may not be compliant with each other. Moreover, Web services use Internet transport protocols (e.g., HTTP, SMTP) and interacting with them requires dealing with limitations of the Internet such access latency, timeouts, lost requests and security issues. These specifics of Web services require special care for supporting dependability of complex distributed systems in integrating them. The provision of effective support for the dependable integration of Web services is still an open issue, which has led to tremendous research effort over the last couple of years, both in industry and academia, as surveyed in the following.

This chapter is organized as follows. Section 2 introduces base Web services architecture, and discusses Web services composition and related dependability requirements. Then, proposed fault tolerance mechanisms for composite Web services are surveyed: Section 3 overviews transaction protocols for the Web based on backward error recovery, and Section 4 presents how forward error recovery can be applied to Web services composition processes. Finally, Section 5 summarizes our analysis, and sketches our current and future work.

2 The Web Services Architecture

Although the definition of the overall Web services architecture is still incomplete, the base standards have already emerged from the W3C⁴, which define a core middleware for Web services, partly building upon results from object-based and component-based middleware technologies. These standards relate to the specification of Web services and of supporting interaction protocols. Furthermore, there already exist various platforms that are compliant with the Web services architecture, including .NET⁵, J2EE⁶ and AXIS⁷. Figure 1 depicts the technology stack of the base Web services architecture, each layer being defined with common protocol choices. Main standards for the Web

⁴ World Wide Web Consortium, <http://www.w3.org>

⁵ Microsoft .NET, <http://www.microsoft.com/net/>.

⁶ Java 2 Platform, Enterprise Edition, <http://java.sun.com/j2ee>.

⁷ Apache AXIS, <http://xml.apache.org/axis>.

services architecture being defined by the W3C Web Service Activity⁸ and the Oasis Consortium⁹ are the following:

- **SOAP** (Simple Object Access Protocol) defines a lightweight protocol for information exchange that sets the rules of how to encode data in XML, and also includes conventions for partly describing the invocation semantics (either synchronous or asynchronous) as well as the SOAP mapping to an Internet transport protocol (e.g., HTTP) [24].
- **WSDL** (Web Services Description Language) is an XML-based language used to specify: (i) the service’s abstract interface that describes the messages exchanged with the service, and (ii) the concrete binding information that contains specific protocol-dependent details including the network end-point address of the service [23].
- **UDDI** (Universal Description, Discovery and Integration) specifies a registry for dynamically locating and advertising Web services [18].

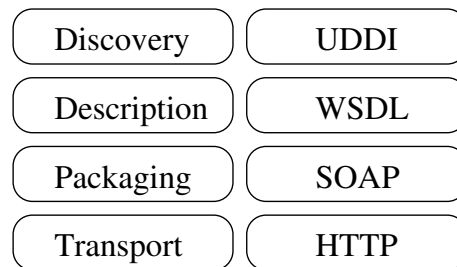


Fig. 1. Web services architecture

Composing Web services then relates to dealing with the assembly of autonomous components so as to deliver a new service out of the components’ primitive services, given the corresponding published interfaces. We use the travel agent case study to illustrate a composite Web service. The travel agent service assists the user in booking complete trips according to his/her requests and is built by composing several existing Web services (e.g., accommodation and flight booking, and car rental Web services) located through a public Web services registry (see Figure 2). Each Web service is an autonomous component, which is not aware of its participation into a composition process. A typical scenario for the travel agent service is as follows:

- The user interacts only with the travel agent whose internal computations are hidden from the user.
- The user sends a query to the travel agent with the date for the travel and the itinerary.

⁸ <http://www.w3c.org/2002/ws/>

⁹ <http://www.oasis-open.org>

- The travel agent proposes to the user complete trips satisfying his/her request, after querying appropriate accommodation, flight and car rental Web services.
- The travel agent makes all the respective bookings and reservations on Web services, according to the user's choices, and returns him/her a confirmation.
- If a full trip cannot be found, the travel agent tries to offer the user some replacement trips but no action can be taken without his/her approval.

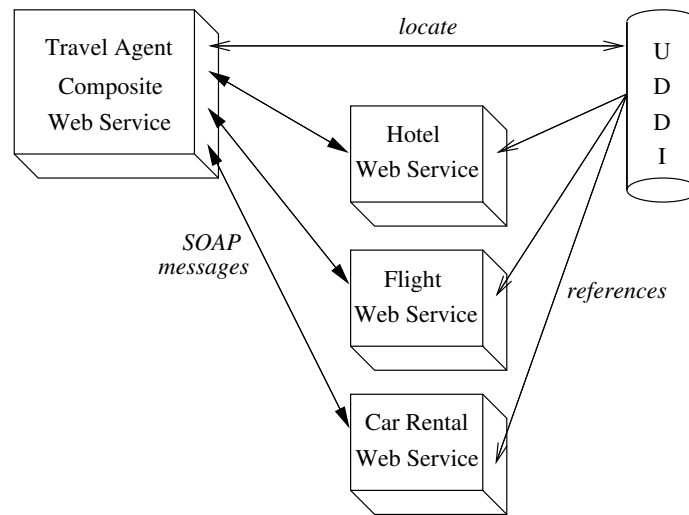


Fig. 2. A composite Web service example: The travel agent service

In the current Web services architecture, interfaces are described in WSDL and published through UDDI. However, supporting composition requires further addressing: (i) the specification of the composition, and (ii) ensuring that the services are composed in a way that guarantees the consistency of both the individual services and the overall composition. This calls for the abstract specification of Web services behaviours (see Section 2.1) and of their composition (see Section 2.2) that allows reasoning about the correctness of interactions with individual Web services. In addition, the composition process must not only define the functional behaviour of the composite service in terms of interactions with the composed services, but also its non functional properties, possibly exploiting middleware-related services (e.g., services relating to WS-Security [14] for enforcing secure interactions). Various non-functional properties (e.g., availability, extendibility, reliability, openness, performance, security, scalability) should be accounted for in the context of Web services. However, enforcing dependability of composite Web services is one of the most challenging issues due to the concern for supporting business processes, combined with the fact that the composition process deals with the assembly of loosely-coupled autonomous components (see Section 2.3).

2.1 Specifying Conversations

To enable a Web service to be correctly integrated in a composition process, Web services conversations (also referred as to choreography) are to be provided in addition to WSDL interfaces for describing the observable behaviour of a Web service by specifying the protocol in terms of message exchanges that should be implemented for correct interaction with the service. As an illustration, a flight booking Web service can publish its behaviour as shown in Figure 3: the booking process starts with the call of the *Login* operation. If the login succeeds, the user can call the *FlightRequest* operation. Then, if the result returns a valid list of flights, the *BookFlight* operation is called by sending the *FlightId* number of the corresponding flight. The conversation ends if (i) the *Login* operation fails, (ii) the user calls *Logout* during the booking process or (iii), the *BookFlight* operation terminates successfully by sending a confirmation message to the user.

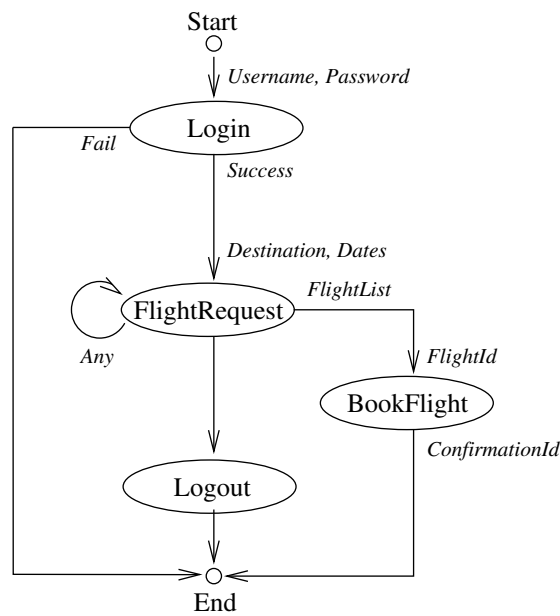


Fig. 3. A conversation for the flight booking Web service

The W3C Choreography Working Group¹⁰ aims at extending the Web services architecture with a standard for specifying Web services conversations. Existing proposals to the W3C for specifying conversations include WSCL (Web Services Conversation Language) [26] and WSCI (Web Service Choreography Interface) [25]. In addition, WSCI can be used to describe additional properties related to the service behaviour (e.g., transactional properties and exceptional behaviour). Note that the specification

¹⁰ <http://www.w3.org/2002/ws/chor/>

of conversations, which enriches the definition of Web services interfaces, should be accounted for, for the definition of Web services registries like UDDI. Instances of Web services should be retrieved with respect to the definition of both the service's interfaces, observable behaviours and non-functional properties (e.g., transactional behaviour). WSDL extensibility elements allow the Web service interface definition to be extended and can be, for example, used to add information describing conversations and other properties. In addition, WSCL can further be published in UDDI registries for retrieving Web services with respect to the conversations that the services support [1].

2.2 Specifying Composition Processes

Web services are integrated according to the specification of a composition process. Such a process may be specified as a graph (or process schema) over the set of composed Web services (see Figure 4). It is worth noting that if some Web services used in a composition process have associated conversation descriptions (e.g., WSCL or WSCI) for the operations they support, the overall composition must conform to all these descriptions. The specification of a composition graph may be:

1. Automatically inferred from the specification of individual services as addressed in [16].
2. Distributed over the specification of the component Web services as in the XL language [7].
3. Given separately with XML-based declarative languages as BPEL, BPML, CSDL and SCSL [11, 2, 4, 28], or in the form of state-charts as undertaken in [6].

The first approach is quite attractive but restricts the composition patterns that may be applied, and cannot thus be used in general. The second approach is the most general, introducing an XML-based programming language. However, this limits the reusability and evolution of (possibly composite) Web services due to the strong coupling of the specification of the composition process with that of the composed services. The third approach directly supports reuse, openness, and evolution of Web services by clearly distinguishing the specification of component Web services (comprising primitive components that are considered as black-box components and/or inner composite components) from the specification of composition. Hence, although there is not yet a consensus about the best approach for specifying composite Web services, it may be anticipated that this will most likely rely on the XML-based specification of a graph over Web services that is decoupled from the specification of the composed Web services. The main reasons that lead to this conclusion include compliance and complementarity with established W3C standards (i.e., WSDL and SOAP), thus providing reusability, openness and extensibility, but also the fact that it is the approach undertaken by most industrial consortia.

2.3 Dependability Requirements

Composite Web services have high dependability requirements that call for dedicated fault tolerance mechanisms due to both the specifics of the Web services architecture

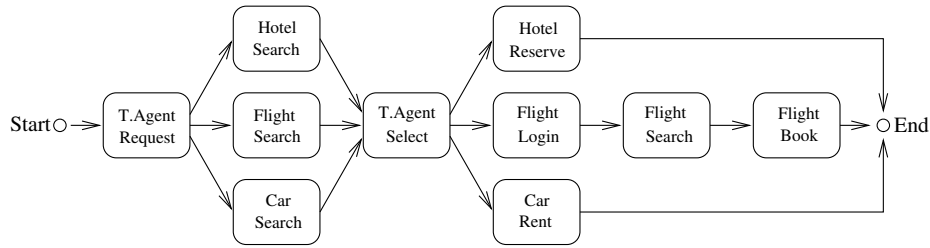


Fig. 4. Composition graph for the travel agent service

and limitations of the Internet, which is not a reliable media [9]. The autonomy of component Web services raises challenging issues in specifying composition processes and in particular exceptional behaviours of composite services when dealing with faults. These faults include but are not limited to (i) faults occurring at the level of the Web services, which may be notified by error messages, (ii) faults at the underlying platform (e.g., hardware faults, timeouts), and (iii) faults due to online upgrades of service components and/or of their interfaces.

In general, the choice of fault tolerance techniques to be exploited for the development of dependable systems depends very much on the fault assumptions and on the system's characteristics and requirements. There are two main classes of error recovery [10]: backward (based on rolling system components back to the previous correct state) and forward error recovery (which involves transforming the system components into any correct state). The former uses either diversely-implemented software or simple retry; the latter is usually application-specific and relies on an exception handling mechanism [5]. It is a widely-accepted fact that the most beneficial way of applying fault tolerance is by associating its measures with system structuring units as this decreases system complexity and makes it easier for developers to apply fault tolerance [20]. Structuring units applied for both building distributed systems and providing their fault tolerance are well-known: they are *distributed transactions* and *atomic actions*¹¹. Distributed transactions [8] use backward error recovery as the main fault tolerance measure in order to satisfy completely or partially the ACID (atomicity, consistency, isolation, durability) properties. Atomic actions [3] allow programmers to apply both backward and forward error recovery. The latter relies on coordinated handling of action exceptions that involves all action participants. Backward error recovery has a limited applicability, and in spite of all its advantages, modern systems are increasingly relying on forward error recovery, which uses appropriate exception handling techniques as a means [5]. Examples of such applications are complex systems involving human beings, COTS components, external devices, several organizations, movement of goods, operations on the environment, real-time systems that do not have time to go back. Integrated Web services clearly fall into this category.

A typical example that shows the need for a specialized fault tolerance mechanism for the Web services architecture is the issue of running distributed transactions over

¹¹ also referred to as conversations, but we will not use this term to avoid confusion with Web services conversations.

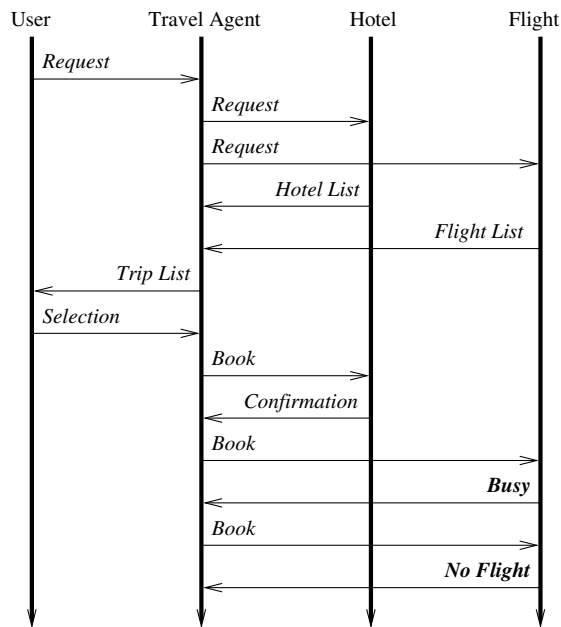


Fig. 5. Sequence diagram for the travel agent service

several autonomous Web services. As an illustration, Figure 5 depicts a sequence diagram example associated with the travel agent service. We consider that for a given trip request (expressed by giving the journey dates and the location), the travel agent finds a list of available hotel rooms and flights to the destination. Then, the user selects a complete trip by choosing a hotel and a flight from the list of trips. Once the user has confirmed the booking, the travel agent makes the hotel reservation, which completes successfully and then, attempts for the flight booking. The flight booking Web service returns an error, informing that the server is busy, and after several attempts, the server returns a *NoFlightAvailable* error message. Several solutions can be applied to handle this type of faults:

1. The hotel reservation can be cancelled –if possible. The user is then informed that this trip is no longer available and he/she can retry booking by selecting another trip (e.g., by changing the dates of the journey).
2. Alternative transport means instead of flight can be proposed to the user (e.g., a train ticket to a close city and renting a car).

These examples reflect two different fault tolerance mechanisms: (i) backward error recovery with cancellation/compensation and retry, and (ii) forward error recovery with an application-specific exception handler that handles the error without necessarily trying to restore the system state back.

Developing fault tolerant mechanisms for composite Web services has been an active area of research over the last couple of years. Existing proposals mainly exploit

backward error recovery, and more specifically, transactions. However, the autonomy of Web services and the Web latency have led to exploit more flexible transactional models and forward error recovery techniques, as discussed in the next two sections.

3 Backward Error Recovery for the Web

Transactions have been proven successful in enforcing dependability in closed distributed systems and are extensively exploited for the implementation of primitive (non-composite) Web services. However, transactions are not suited for making the composition of Web services fault tolerant in general, for at least two reasons:

- The management of transactions that are distributed over Web services requires cooperation among the transactional supports of individual Web services, which may not be compliant with each other and may not be willing to do so given their intrinsic autonomy and the fact that they span different administrative domains.
- Locking resources (i.e., the Web service itself in the most general case) until the termination of the embedding transaction is in general not appropriate for Web services, still due to their autonomy, and also to the fact that they potentially have a large number of concurrent clients that will not stand extensive delays.

Enhanced transactional models have been considered to alleviate the latter shortcoming. In particular, the split model (also referred to as open-nested transactions) where transactions may split into a number of concurrent sub-transactions that can commit independently allows reduction of the latency due to locking [19]. Typically, sub-transactions are matched to the transactions already supported by Web services (e.g., transactional booking offered by a service). Hence, transactions over composite services do not increase the access latency as offered by the individual services. Enforcing the atomicity property over a transaction that has been split into a number of sub-transactions then requires using compensation over committed sub-transactions in the case of transaction abortion. However, to support this, Web services should provide compensating operations for all the operations they offer. Such an issue is in particular addressed by the BPEL [11] and WSCI [26] languages for specifying composite services, which allow compensating operations associated with the services operations to be defined. It is worth noting that using compensation for aborting distributed transactions must extend to all the participating Web services (i.e., cascading compensation by analogy with cascading abort). Such a concern is addressed in [15]. This paper introduces a middleware whose API may be exploited by clients of a composite service for specifying and executing a (open-nested) transaction over a set of Web services whose termination is dictated by the outcomes of the transactional operations invoked on the individual services. In addition to client-side solutions to the coordination of distributed open-nested transactions, work is undertaken in the area of distributed transaction protocols supporting the deployment of transactions over the Web, while not imposing long-lived locks over Web resources. We discuss here the two main proposals aimed at the Web services architecture: (i) the Business Transaction Protocol (BTP)[17], and (ii) Web Services Transaction (WS-Transaction) [13].

BTP introduces two different transaction models for the Web: (i) the *atomic business transactions* (or *atoms*), and (ii) the *cohesive business transactions* (or *cohesions*). A composite application can be built from both *atoms* and *cohesions* that can be nested. In the *atomic business transaction* model, several processes are executed within a transaction and either all complete or all fail. This is similar to distributed ACID transactions on tightly coupled systems. However, the isolation property is relaxed and intermediate committed values can be seen by external systems (i.e., systems not enrolled in the transaction). Figure 6 illustrates the *atomic business transaction* model using the travel agent service involving a flight booking Web service (*Flight*) and an accommodation booking Web service (*Hotel*). In this scenario, the hotel room booking fails while the flight booking succeeds, which leads to cancellation of the booked flight before the end of the transaction:

1. *Travel Agent* sends the request messages to *Flight* and to *Hotel* Web services.
2. *Flight* and *Hotel* respond (*Confirm* messages) with listings of available flights and hotel rooms.
3. *Travel Agent* orders the bookings by initiating commitments (*Prepare* messages).
4. *Flight* Web service returns *Prepared* and is ready to commit, while the *Hotel* Web service returns a *Fail* error message. Commit is no longer possible on the *Hotel* Web service for this transaction.
5. *Travel Agent* cancels the transaction on the *Flight* Web service by sending the *Cancel* order.
6. *Flight* Web service confirms cancellation with the *Cancelled* message.

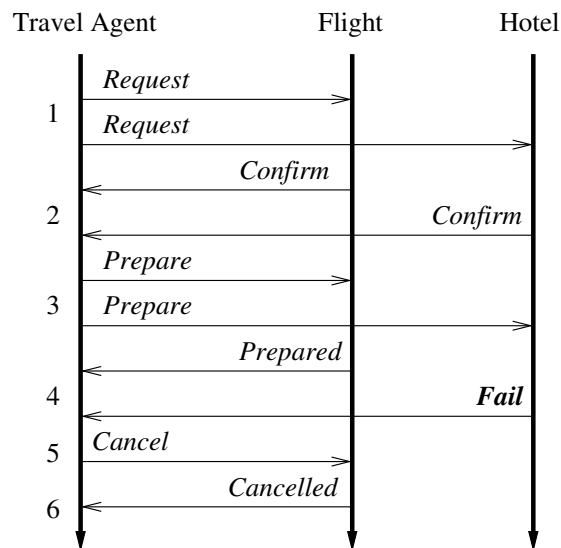


Fig. 6. BTP Atomic Business Transaction

The *cohesive business transaction* model allows non-ACID transactions to be defined by not requiring successful termination of all the transaction's participants for committing. A travel agent service scenario example for illustrating *cohesive business transactions* is given in Figure 7, where the flight booking is performed on two distinct Web services. In this example, the transaction, which was originally initiated with three participants, ends with two commits and one abortion:

1. *Travel Agent* sends the request messages to the two flight booking Web services, *Air France* and *British Airways* and to the *Hotel* Web service.
2. Web services return response messages to the *Travel Agent*.
3. *Travel Agent* selects *Air France* for the flight booking, and therefore sends a *Cancel* message to *British Airways* Web service and a *Prepare* message to the two other Web services.
4. *Air France* and *Hotel* Web services acknowledge with the *Prepared* message and *British Airways* confirms the cancellation with the *Cancelled* message.
5. *Travel Agent* confirms commits (*Confirm* messages).
6. Web services acknowledge (*Confirmed* messages).

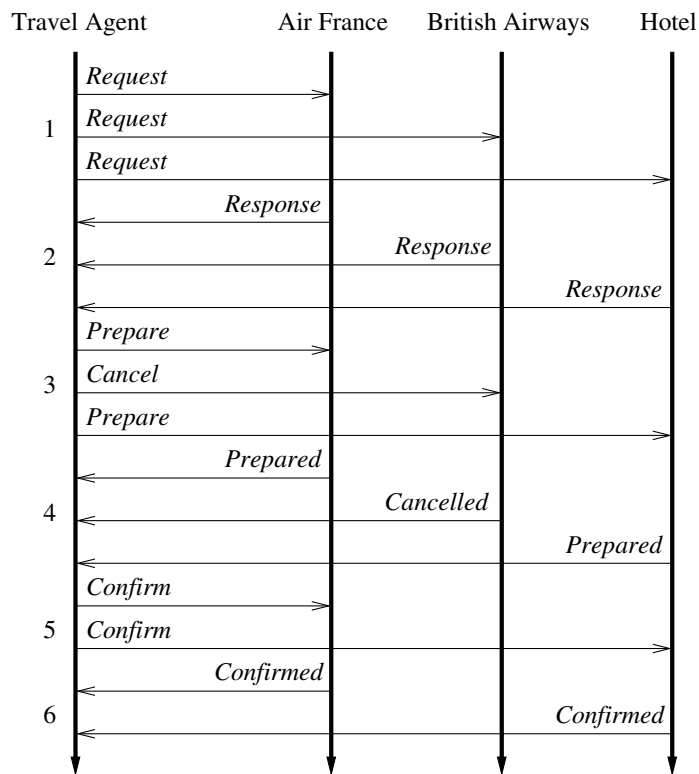


Fig. 7. BTP Cohesive Business Transaction

WS-Transaction [13] defines a specialization of WS-Coordination [12], which is an extensible framework for specifying distributed protocols that coordinate the execution of Web services, and that can be used in conjunction with BPEL. Like BTP, it offers two different transaction models: (i) *atomic transactions* (AT) and (ii) *business activity* (BA). An *atomic transaction* adheres to the traditional ACID properties with a two-phase commit protocol. Note that as opposed to the BTP *atomic business transactions*, the isolation property is not relaxed in WS-Transactions, which as we mentioned before, is not suitable for the majority of Web service applications. The *business activity* protocol specifically serves coordinating the execution of open-nested transactions over a set of activities, through a coordinator activity. If there is a need for a coordinated activity to be compensated, the coordinator sends *compensate* messages to all the participants involved in the activity. Then, each participant replies by sending back either a *compensated* or a *faulted* message, depending on whether the required compensation operation was successfully completed or not. However, there is no requirement for an agreement on the outcome, and any participant can leave the coordinated activity in which it is engaged, prior to the termination of peer participants. A WS-Transaction *business activity* example is shown in Figure 8, with an *Airline* Web service and an *Hotel* Web service:

1. The *Travel Agent* initiates the *Business Activity* with the *Flight* and *Hotel* participants by sending the *Request* messages.
2. The *Flight* and *Hotel* Web services enroll in the transaction (*Register* messages) by returning list of respective availabilities.
3. *Travel Agent* initiate booking on the *Flight* Web service (*Complete* message).
4. The *Flight* Web service returns *Completed* to confirm commitment, while the *Hotel* Web service returns an error message *Faulted* and can no longer commit the transaction.
5. In order to abort the whole transaction and restore the state, the *Travel Agent* sends a *Compensate* message to the *Flight* Web service which has already completed the (sub)-transaction and a *Forget* message to the *Hotel* Web service.
6. The *Flight* Web service compensates the committed transaction by cancelling the booking order and confirms with the *Compensated* message sent back to the *Travel Agent*.
7. If the *Flight* Web service cannot compensate the booked operation, it returns an error message *Faulted* back to the *Travel Agent*.
8. In this case, the *Travel Agent* sends a *Forget* message to the *Flight* Web service. The flight has been booked and cannot be cancelled.

Although there is not yet a consensus on a standard protocol for managing transactions on the Web, various implementations of these protocols are already available: the JOTM transaction manager¹² and Cohesions¹³ implement the BTP protocol, and the Collaxa Orchestration Server¹⁴ permits the use of the WS-Transaction protocol when composing Web services with the BPEL language. However, solutions to the dependable composition of Web services that use primarily transactions do not cope with all

¹² <http://www.objectweb.org/jotm/>

¹³ <http://www.choreology.com/>

¹⁴ <http://www.collaxa.com/>

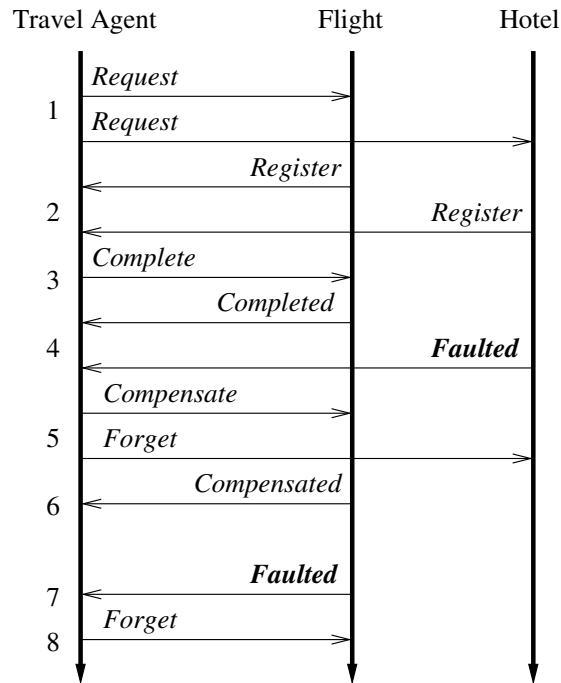


Fig. 8. WS-Transaction Business Activity

the specifics of Web services. A major source of penalty lies in the use of backward error recovery in an open system such as the Internet, which is mainly oriented towards tolerating hardware faults but poorly suited to the deployment of cooperation-based mechanisms over autonomous component systems that often require cooperative application-level exception handling among component systems. Even with the use of *cohesions*, which does not necessarily roll back all participant states in case of abortion of one of them, they do not specify any specific way of providing fault tolerance, so everything is left to programmers. Moreover, cancellation or compensation does not always work in many real-life situations, which involve documents, goods, money as well as humans (clients, operators, managers, etc.) and which require application-specific error handling.

4 Forward Error Recovery for the Web

Forward error recovery, using an exception handling mechanism is extensively exploited in the specifications of composite Web services in order to handle error occurrences (e.g., [11], [2], [25]). For instance, in BPEL, exception handlers (referred to as fault handlers) can be associated to a (possibly nested) activity so that when an error occurs inside an activity, its execution terminates, and the corresponding exception handler is executed. However, when an activity is defined as a concurrent process and at

least one embedded activity signals an exception, all the embedded activities are terminated as soon as one signaled exception is caught, and only the handler for this specific exception is executed. Hence, error recovery actually accounts for a single exception and thus cannot ensure recovery of a correct state. The only case where correct state recovery may be ensured is when the effect of all the aborted activities are rolled back to a previous state, which may not be supported in general, in the context of Web services, as discussed previously. The shortcoming of BPEL actually applies to all XML-based languages for Web services composition that integrate support for specifying concurrent activities and exception handling.

A solution to the above issue lies in structuring the composition of Web services in terms of *coordinated atomic actions*. The Coordinated Atomic Action (or CA action) concept [27] is a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting components. Atomic actions are used to control cooperative concurrency and to implement coordinated error recovery whilst ACID transactions are used to maintain the consistency of shared resources. A CA action is designed as a set of participants cooperating inside it and a set of resources accessed by them. In the course of the action, participants can access resources that have ACID properties. Action participants either reach the end of the action and produce a normal outcome or, if one or more exceptions are raised, they all are involved in their coordinated handling. If several exceptions have been raised concurrently, they are resolved [3] using a resolution tree imposing a partial order on all action exceptions, and the participants handle the resolved exception. If this handling is successful the action completes normally, but if handling is not possible then all responsibility for recovery is passed to the containing action where an external action exception is propagated. CA actions provide a base structuring mechanism for developing fault tolerant composite Web services: a CA action specifies the collaborative realization of a given function by composed services, and Web services correspond to external resources. However, as for transactions, ACID properties over external resources are not suited in the case of Web services. We have therefore introduced the notion of Web Services Composition Action (WSCA) that differs from CA actions in relaxing the transactional requirements over external resources (which are not suitable for wide-area open systems) and the introduction of dynamic nesting of WSCAs (i.e., nested calls of WSCAs for the sake of modularity) [22].

The travel agent service is used to illustrate how WSCAs can be applied for specifying the composition of Web services. We consider joint booking of accommodation and flights using separate hotel and airline Web services. Then, the composed Web service's operation is specified using WSCAs as follows. The top-level *TravelAgent* WSCA comprises the *User* and the *Travel* participants; the former interacts with the user while the latter achieves joint booking according to the user's request through call to the WSCA that composes the *Flight* and the *Hotel* participants¹⁵. Figure 9 depicts the software architecture of the travel agent service where rectangles represents components which

¹⁵ Such a workflow process is certainly not the most common since the user is in general requested for confirmation prior to booking. However, this scenario that applies most certainly to in-hurry-not-bother users enables concise illustration of the various recovery schemes that are supported.

can be either WSCA participants or Web services, and elliptical nodes represents connectors such as WSCAs connectors and SOAP connectors.

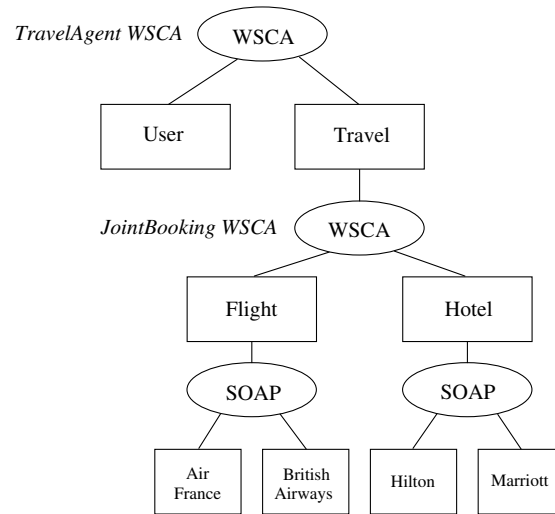


Fig. 9. WSCA architecture

A diagrammatic specification of the WSCAs is shown in Figure 10. In *TravelAgent*, the *User* participant requests the *Travel* participant to book a flight ticket and a hotel room for the duration of the given stay. This leads the *Travel* participant to invoke the *JointBooking* WSCA that composes the *Hotel* Web service and the *Airline* Web service. The participants of the *JointBooking* WSCA respectively requests for a hotel room and a flight ticket, given the destination and departure and return dates provided by the user. Each request is subdivided into reservation for the given period and subsequent booking if the reservation succeeds¹⁶. In the case where either the reservation or the booking fails, the participant raises the *unavailable* exception that is cooperatively handled at the level of the *JointBooking* WSCA denoted by the greyed box in the figure. If both participants signal the *unavailable* exception, then *Travel* signals the *abort* exception so that the exception gets handled by *TravelAgent* in a cooperation with the *User* (e.g., by choosing an alternative date). If only one participant raises the *unavailable* exception, cooperative exception handling includes an attempt by the other participant to find an alternative booking. If this retry fails, the booking that has succeeded is cancelled and the abort exception is signaled to the calling *TravelAgent* WSCA for recovery with user intervention.

¹⁶ We assume here that all Web services understand and use term *unavailable* in the same way. To deal with the problem of meanings in a general fashion one could apply techniques that use ontology and standard ways of representing semantic information. Such an ontology for travel business partners can be found on <http://opentravel.org>.

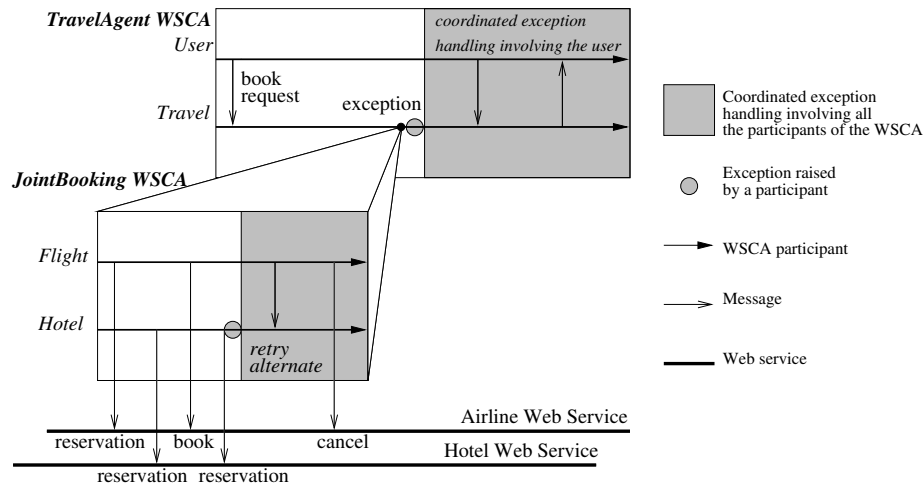


Fig. 10. WSCA for composing Web services

Compared to the solutions that introduce transactional supports for composed Web services, using WSCAs mainly differs in that it exploits forward error recovery at the composition level, while enabling exploitation of transactional supports offered by individual Web services – if available. Hence, the underlying protocol for interaction among Web services remains the one of the Web services architecture (i.e., SOAP) and does not need to be complemented with a distributed transaction protocol. Similarly to this solution, the one of [15] does not require any new protocol to support distributed open-nested transactions. An open-nested transaction is declared on the client side by grouping transactions of the individual Web services, through call to a dedicated function of the middleware running on the client. The transaction then gets aborted by the middleware using compensation operations offered by the individual Web services, according to conditions set by the client over the outcomes of the grouped transactions. The solution offered by WSCA is then more general since it allows forward error recovery involving several services to be specified at the composition level, enabling in particular to integrate non-transactional Web services while still enforcing dependability of the composite service and partial results of a nested action to be reported to the higher level action.

There is a number of Java and Ada implementations of CA actions developed for different platforms, environments and applications. A complete RMI Java framework was developed several years ago [29] and since then it has been applied in a number of industry-oriented case studies. It offers a number of classes (for defining actions, action participants, exception handlers) and a runtime support in a form of the action manager object. Recently it has been used for a preliminary experimental work on implementing a prototype Travel Agent system [21]. A Java-based local runtime support for WSCA is under development now. It is built as an adaptation of this extended CA action Java framework.

5 Conclusion

The Web services architecture is expected to play a major role in developing next generation distributed systems. However, the architecture needs to evolve to support all the requirements appertained to distributed systems. Addressing such requirements relates, in particular, in reusing solutions from the distributed system community. However, most solutions will not be reusable as is, mainly because of the openness of the Internet. Hence, making evolve the Web services architecture to support the thorough development of distributed systems raises a number of challenges.

This paper has addressed one of the issues raised in this context, which is the dependable composition of Web services, i.e., understanding how fault tolerance should be addressed in the Web services architecture. While dependability in closed distributed systems is conveniently addressed by transactions when concerned with both concurrency control and failure occurrences, it can hardly rely on such a mechanism in an open environment. A major source of difficulty lies in the use of backward error recovery which is mainly oriented towards tolerating hardware faults but poorly suited to the deployment of cooperation-based mechanisms over autonomous component systems that often require cooperative application-level exception handling among component systems. One solution to this concern lies in forward error recovery that enables accounting for the specific of Web services and that leads to structure Web services-based systems in terms of co-operative actions. In particular, we are able to address dependable service composition in a way that neither undermines the Web service's autonomy nor increases their individual access latency.

We are currently working on the formal specification of WSCAs for enabling rigorous reasoning about the behavior of composite Web services regarding both the correctness of the composition and offered dependability properties. The specification of composite Web services allows carrying out a number of analyses with respect to the correctness and the dependable behavior of composite services. Except classical static type checking, the correctness of the composite service may be checked statically with respect to the usage of individual services. In addition, the same specification can be used for implementing executable assertions to check the composite service behaviour online. Reasoning about the dependable behaviour of composite Web services lies in the precise characterization of the dependability properties that hold over the states of the individual Web services after the execution of WSCAs. We are in particular interested in the specification of properties relating to the relaxed form of atomicity that is introduced by the exploitation of open-nested transactions within WSCA.

We are further implementing a base middleware support for WSCAs. The middleware includes the generation of composite Web services from WSCA specifications and a service for locating Web services that implements behavioral specification matching. In addition, we target development of related middleware support for improving the overall quality of composite Web services. We are in particular interested in developing a specialized caching support intended for reducing response time.

Acknowledgments

We would like to thank Marie-Claude Gaudel and Brian Randell for their helpful comments and discussions. This research is partially supported by the European IST DSoS (Dependable Systems of Systems) project (IST-1999-11585)¹⁷.

References

1. D. Beringer, H. Kuno, and M. Lemon. Using WSCL in a UDDI Registry 1.02, 2001. UDDI Working Draft Technical Note Document, http://www.uddi.org/pubs/wscl_TN_forUDDI_5_16_011.pdf.
2. BPMI.org. Business Process Modeling Language (BPML), Version 1.0, 2002. <http://www.bpmi.org/bpml.esp>.
3. R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8), 1986.
4. F. Casati, M. Sayal, and M-C. Shan. Developing e-services for composing e-services. In *Proceedings of CAISE'2001, LNCS 2068*, 2001.
5. F. Cristian. *Dependability of Resilient Computers*, chapter Exception Handling, pages 68–97. Blackwell Scientific Publications, 1989.
6. M-C. Fauvet, M. Dumas, B. Benatallah, and H-Y. Paik. Peer-to-peer traced execution of composite services. In *Proceedings of TES'2001, LNCS 2193*, 2001.
7. D. Florescu, A. Grunhagen, and D. Kossmann. XL: An XML language for Web service specification and composition. In *Proceedings of the WWW'02 Conference*, 2002.
8. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
9. M. Kalyanakrishnan, R.K. Iyer, and J.U. Patel. Reliability of Internet hosts: a case study from the end user's perspective. *Computer Networks*, (31):47–57, 1999.
10. P. A. Lee and T. Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer - Verlag, 2nd edition, 1990.
11. Microsoft and BEA and IBM. Business Process Execution Language for Web Services (BPEL4WS), Version 1.0, 2002. <http://www.ibm.com/developerworks/library/ws-bpel/>.
12. Microsoft and BEA and IBM. Web Services Coordination (WS-Coordination), 2002. <http://www.ibm.com/developerworks/library/ws-coor/>.
13. Microsoft and BEA and IBM. Web Services Transaction (WS-Transaction), 2002. <http://www.ibm.com/developerworks/library/ws-transpec/>.
14. Microsoft and IBM and VeriSign. Web Services Security (WS-Security), Version 1.0, 2002. <http://www.ibm.com/developerworks/library/ws-secure/>.
15. T. Mikalsen, S. Tai, and I. Rouvellou. Transactional attitudes: Reliable composition of autonomous Web services. In *DSN 2002, Workshop on Dependable Middleware-based Systems (WDMS 2002)*, 2002.
16. S. Narayanan and S. McIlraith. Simulation, verification and automated composition of Web services. In *Proceedings of the WWW'02 Conference*, 2002.
17. Oasis Committee. Business Transaction Protocol (BTP), Version 1.0, 2002. <http://www.oasis-open.org/committees/business-transactions/>.
18. Oasis Committee. Universal Description, Discovery and Integration (UDDI), Version 3 Specification, 2002. <http://www.uddi.org>.

¹⁷ <http://www.newcastle.research.ec.org/dsos/>

19. C. Pu, G. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. *Proceedings of the 14th VLDB Conference*, 1988.
20. B. Randell. Recursive structured distributed computing systems. In *Proc. of the 3rd Symp. on Reliability in Distributed Software and Database Systems*, pages 3–11, Florida, USA, 1983.
21. A. Romanovsky, P. Periorellis, and A.F. Zorzo. Structuring integrated Web applications for fault tolerance, 2003. To be presented at the 6th Int. Symposium on Autonomous Decentralised Systems. Pisa, Italy, April 2003. (a preliminary version: Technical Report CS-TR-765, University of Newcastle upon Tyne).
22. F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Dependability in the web service architecture. In *Proceedings of the ICSE Workshop on Architecting Dependable Systems*, Orlando, USA, May 2002.
23. W3C. Web Services Description Language (WSDL) 1.1, W3C Note, 2001. <http://www.w3.org/TR/wsdl> (W3C Working Draft for version 1.2 is available at <http://www.w3.org/TR/wsdl12>).
24. W3C. Simple Object Access Protocol (SOAP) 1.2, W3C Candidate Recommendation, 2002. <http://www.w3.org/TR/soap12-part0/>.
25. W3C. Web Service Choreography Interface (WSCI) 1.0, W3C Note, 2002. <http://www.w3.org/TR/wsci/>.
26. W3C. Web Services Conversation Language (WSCL) 1.0, W3C Note, 2002. <http://www.w3.org/TR/wscl10/>.
27. J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the Twenty-Fifth IEEE International Symposium on Fault-Tolerant Computing*, 1995.
28. J. Yang and P. Papazoglou. Web component: A substrate for Web service reuse and composition. In *Proceedings of CAISE'2002*, 2002.
29. A.F. Zorzo and R. J. Stroud. An object-oriented framework for dependable multiparty interactions. In *proc. of Conf. on Object-Oriented Programming Systems and Applications (OOPSLA'99)*, *ACM Sigplan Notices 34(10)*, pages 435–446, 1999.