Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 302 (2014) 5–28

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Reliable Composite Web Services Execution: Towards a Dynamic Recovery Decision

Rafael Angarita [1]

*PSL, Université Paris-Dauphine*
*75775 Paris Cedex 16*
*France CNRS, LAMSADE UMR 7243*

Yudith Cardinale[2]

*Departamento de Computación*
*Universidad Simón Bolívar*
*Caracas, Venezuela*

Marta Rukoz [3]

*PSL, Université Paris-Dauphine*
*75775 Paris Cedex 16*
*France CNRS, LAMSADE UMR 7243*
*Université Paris Ouest Nanterre La Défense*
*France*

**Abstract**

During the execution of a Composite Web Service (CWS), different faults may occur that cause WSs failures. There exist strategies that can be applied to repair these failures, such as: WS retry, WS substitution, compensation, roll-back, or replication. Each strategy has advantages and disadvantages on different execution scenarios and can produce different impact on the CWS *QoS*. Hence, it is important to define a dynamic fault tolerant strategy which takes into account environment and execution information to accordingly decide the appropriate recovery strategy. We present a preliminary study in order to analyze the impact on the CWS total execution time of different recovery strategies on different scenarios. The experimental results show that under different conditions, recovery strategies behave differently. This analysis represents a first step towards the definition of a model to dynamically decide which recovery strategy is the best choice by taking into account the context-information when the failure occurs.

*Keywords:* Fault-tolerance, dynamic, execution, context-aware.

[1] Email: rafael.angarita@lamsade.dauphine.fr

[2] Email: yudith@ldc.usb.ve

[3] Email: marta.rukoz@lamsade.dauphine.fr

http://dx.doi.org/10.1016/j.entcs.2014.01.018

# 1 Introduction

Nowadays, SOA architecture is used as a platform for business applications for accessing data and services in distributed environments. The constantly increasing number of such applications currently deployed over the Internet is enabled by the latest SOA techniques, such as Web Services (WS) and Web 3.0, and is a consequence of the need for business integration and collaboration. With machine intelligence, users can resolve complex problems that require the interaction among different tasks. One of the major goals of the Web 3.0 is to support automatic and transparent WS composition and execution, allowing a complex user request to be satisfied by a Composite Web Service (CWS). Hence, in a CWS, functionalities of individual WSs (possibly from different providers) are combined to resolve the complex query [2].

Most of the generic or domain-tailored solutions for creating, executing, and managing such CWSs have been extensively treated in the literature, exhibiting sophisticated interfaces and a multitude of connectors to subsystems to represent functional properties, and increasing support for non-functional properties [5,3,13,17,11,6,15]. Nevertheless, recovery of failures for reliable execution have received relatively limited attention.

During the execution of a CWS, different faults may occur that cause a WS failure. However, a fault-tolerant CWS is the one that, upon a service failure, ends up the whole composite service (e.g., by retrying, substituting, or replicating the faulty WS) or leaves the execution in a safe state (e.g., by rollbacking or compensating the faulty WS and the related executed WSs). In this sense, fault tolerant CWS becomes a key mechanism to cope with challenges of open-world applications in dynamic changing and untrusted operating environments to ensure that the whole system remains in a consistent state even in the presence of failures [23].

Several techniques have been proposed to implement fault tolerant CWS execution. In some works, transactional properties of component WSs (e.g., retriable, compensable or not) are considered to ensure the classical ACID (all-or-nothing) properties in CWSs [13,11,6,15,9,4]. In this context, failures during the execution of a CWS can be repaired by backward or forward recovery processes. Backward recovery implies to undo the work done until the failure and go back to the initial consistent state (before the execution started), by roll-back or compensation techniques. Forward recovery tries to repair the failure and continue the execution, using retry and substitution, for example. In previous works, we presented our solutions based on backward and forward recovery [8,10].

However, backward recovery means that users do not get the desired answer to their queries, besides roll-back techniques that claim for logs in persistent storage to enable recovery after a re-start, reboot, or crash. The need for synchronous logging slows down the execution speed during normal operation and the reliability of these mechanisms depends on the reliability of the storage. Forward recovery could imply long waiting times, due of the invested time to repair failures until users finally get the response.

Others works consider replication of WSs instead of transactional properties to provide forward recovery. With this strategy, several equivalent WSs are simultaneously invoked and the response is taken from the first successfully finished one [28,1]. Because WSs can be created and updated on-the-fly, the execution system needs to dynamically detect changes during run-time, and adapt execution to the availability of the existing WSs. Replicating a service creates the need for mechanisms to distribute messages, order requests, and coordinate replicas. However, sometimes it is not possible to replicate the invocation of several equivalent WSs or substitute the faulty WS for an equivalent one, because equivalent WSs are not available or because the extra consumption of resources.

Each strategy has advantages and disadvantages on different scenarios. These scenarios can be defined by, for example, the execution state at the moment of the failure (e.g., how many WSs have been successfully executed, how many WSs have not been invoked), the impact of the recovery strategy in the QoS of the CWS, and the stability and reliability of the system. Are all recovery techniques equally practical, effective, and efficient? When is it better to apply backward or forward recovery? Is replication the best strategy? Some replicas may become unavailable permanently, while some new replicas may join in. WSs may be updated without any notification, and the Internet traffic load and server workload are also changing from time to time. Persistent storage can be available or not. These unpredictable characteristics of WS environments provide a challenge for optimal fault tolerance strategy determination. There is an urgent need for more general and smarter fault tolerance strategies, which are context-information aware and can be dynamically and automatically reconfigured for meeting different user requirements and changing environments.

Focused on that need, in this paper we present a study to analyze the impact on the CWS total execution time of different recovery strategies in different scenarios. We focus on backward, forward, and replication recovery techniques. The experimental results show that recovery strategies behave differently under different conditions. This analysis represents a first step towards the definition of a model to dynamically decide which recovery strategy is the best choice taking into account the execution state when a failure occurs, context-information, and the impact on the $QoS$.

The remainder of this paper is organized as follows. Section 2 briefly describes the most important concepts related to fault tolerance in CWS. In Section 3 we characterize the CWS and the environment in terms of execution time a the basis of our study. The experimental study is presented in Section 4. Section 5 presents relevant related works. Finally, our conclusions and future work are presented in Section 6.

# 2 Fault Tolerance for Composite Web Services: the background

This section recalls some important concepts regarding Composite Web Service (CWS) execution and the most important recovery approaches implemented in this area. We also describe a classification of failures that can affect the CWS execution and point out which are considered in this work.

## *2.1 Composite Web Service*

A Composite Web Service, denoted as CWS, is a combination of several WSs to produce more complex services that satisfy more complex user requests. It concerns *which* and *how* WSs are combined to obtain the desired results. A CWS can be represented in structures such as workflows, graphs, or Petri Nets indicating, for example, the control flow, data flow, WSs execution order, and/or WSs behavior. The structure representing a CWS can be manually or automatically generated. Users can manually specify *how* functionality of WSs are combined or a "composer agent" can automatically decide *which* and *how* WSs are combined, according the desired query. In both cases, the execution of a CWS is carried out by an "execution engine" that invokes the WSs respecting such structure. In this paper, we consider a CWS represented by a graph, formally defined in Def. 2.1.

***Definition 2.1*** **Composite Web Service Graph.** A Composite Web Service Graph, denoted as $G = (V, E)$, is a directed acyclic graph with the following considerations:

- Nodes in $V$ represent the component WSs, such that $V = \{ws_i, i = 1..m\}$ and $ws_i$ is a component WS.
- Arcs in $E$ denote the execution flow among component WSs. Execution flow is defined by data or control flow relationships between two services. Data flow relationship is defined in terms of service input/output attributes, such that output values produced by a WS are part of the input parameters of another WS. Control flow relationship is defined by execution order restrictions (e.g., business process order, transactional property, concurrence control, deadlock avoidance) that dictate that a WS has to be executed after another WS finishes its execution. Control flow can be designated by control signals (as well called control data). Thus, if $ws_i, ws_j \in V$ and $(ws_i, ws_j) \in E$, then $ws_i$ produces output attributes from which at least one is an input parameter of service $ws_j$ and/or $ws_j$ has to wait for a control signal from $ws_i$. In other words, if $O(ws_i)$ represents the set of output attributes and control signals that $ws_i$ produces and $I(ws_j)$ represents the set of input parameters and control signals needed to invoke $ws_j$, then $O(ws_i) \cap I(ws_j) \neq \emptyset$.
- Entry nodes represent WSs whose input attributes are provided by the user, then $\exists\ ws_i \in V : I(ws_i) = \emptyset$.
- Output nodes represent WSs that produce the final desired output attributes to

the user, then $\exists\ ws_i \in V : O(ws_i) = \emptyset$.

Note that workflows, bipartite graphs, and Petri Nets, the most popular structures used to represent CWS, can be matched to our graph definition.

### *2.2* CWS *control execution*

The execution of a CWS implies the invocation of all component WSs according to the execution flow imposed by the structure representing the CWS (Def. 2.1). Thus, there exist two basic variants of execution scenarios. In a sequential scenario, some WSs cannot be invoked until previous services have finished, because they work on the results of previous services or some control restrictions impose sequentiality. In a parallel scenario, several services can be invoked simultaneously because they do not have data or control flow dependencies.

The execution control can be centralized or distributed. Centralized approaches consider a coordinator managing the whole execution process [20,26]. In distributed approaches, the execution process proceeds with collaboration of several participants without a central coordinator [4,1]. On the other hand, the execution control could be attached to the WS [13,15] or it could be independent of its implementation [9]. Some execution engines, such as the IBM framework BPWS4J [4] or the open source Orchestra [5] solutions actually execute CWS specified with BPEL4WS in a centralized fashion and the execution control is attached to WSs.

In this work, we consider distributed execution engine and execution control independent of WSs implementation.

### *2.3 Failures classification*

During the execution of a CWS, failures can occur at multiple levels: hardware, operating system, web services, execution engine, and network. These failures result in reduced performance and can cause different behaviors in the execution. According the nature of failures, we divide faults into two types:

- Silent faults. These kind of faults are generic to all WSs and cause WSs to not respond because they are not available or a crash occurred in the platform. Some examples of silent faults are (i) communication timeout, (ii) service unavailable, (iii) bad gateway, and (iv) server error. Silent faults can be easily identified by the execution engine.
- Logic faults. Logic faults are specific to different WSs and are caused by error in inputs attributes (e.g., bad format, out of valid range, calculation faults) and byzantine faults (the WS still responds to invocation but in a wrong way). Also, various exceptions thrown by the WS to the service users are classified into the logic-related faults. It is difficult for the execution engine to identify such type of faults.

[4] Business Process for Web Services JavaTM, `http://www.alphaworks.ibm.com/tech/bpws4j` - Extracted on April 2013

[5] Orchestra,`http://orchestra.ow2.org/xwiki/bin/view/M ain/WebHome` - Extracted on April 2013

Regarding faults, in this paper we consider: (i) the execution engines run far from WS hosts, in reliable servers such as clusters computing, they do not fail, their data network is highly secure, and they are not affected by WS faults (the execution control is detached from WSs); and (ii) component WSs can suffer silent failures; run-time failures caused by logic faults are not considered.

### *2.4 Fault tolerant* CWS *execution*

Some execution engines are capable to manage failures during the execution. Ones are based on exception handling[21,18], others are based on transactional properties [11,4,7], some others use a combination of both approaches [13,15], and others base the fault tolerance on replication techniques [28,1].

In previous researches in the field of supporting reliability and fault tolerance in WS composition, only low level programming constructs such as exception handling (for example in WSBPEL) were considered. Exception handling normally is explicitly specified at design time, regarding how exceptions are handled and specifying the behavior of the CWS when an exception is thrown. This approach is normally used to manage logic faults, which are specific to WSs.

More recently the reliability and fault tolerance for CWS have been handled at a higher level of abstraction, i.e., at the execution flow structure level such as workflows or graphs. Therefore, technology independent methods for fault-tolerant CWS have emerged, such as transactional properties and replication.

Transactional properties implicitly describe the behavior in case of failures and are considered to ensure the classical ACID (all-or-nothing) transactional properties. When transactional properties are not considered, the system consistence is responsibility of users/designers.

WSs that provide transactional properties are useful to guarantee reliable CWSs execution and to ensure the whole system consistent state even in presence of failures. The basic recovery techniques supported by transactional properties are:

- Backward recovery: it consists in restoring the state that the system had at the beginning of the CWS execution; i.e., all the effects produced by the failed WS and the previous executed WSs before the failure are semantically undone by roll-back or compensation techniques.
- Forward recovery: it consists in repairing the failure to allow the failed WS to continue its execution; retry and substitution are some techniques used to provide forward recovery; with the popularization of WSs, more and more functionally equivalent WSs are diversely designed and developed by different organizations, making WS substitution an attractive fault tolerant choice for service reliability improvement.

Figures 1 and 2 graphically show the different recovery techniques for a simple WS and for a CWS based on transactional properties.

Backward recovery means that users do not get the desired answer to their queries. Moreover, this strategy may imply roll-back techniques, which claim for logs in persistent storage to enable recovery after a re-start, reboot, or crash. The need

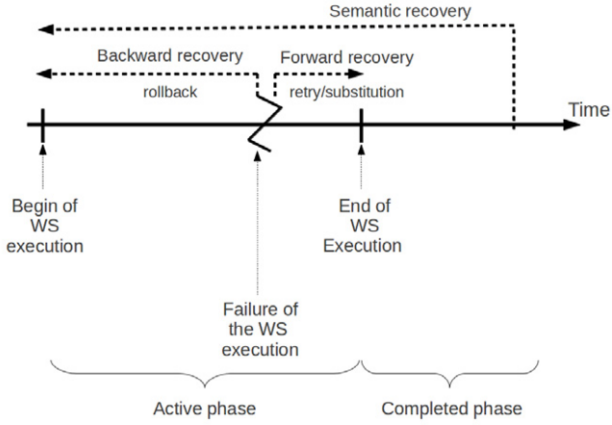



Fig. 1. Lifecycle of a WS and the different possible recovery techniques

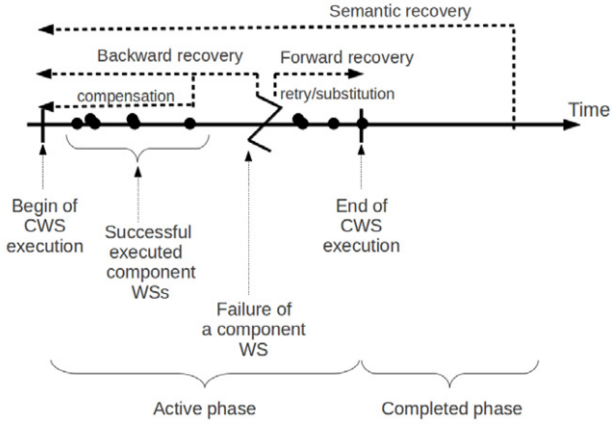



Fig. 2. Lifecycle of a CWS and the different possible recovery techniques

for synchronous logging slows down the execution speed during normal operation and the reliability of these mechanisms depends on the reliability of the storage. Forward recovery could imply long waiting times because of the invested time to repair failures until the users finally get response. They can be difficult to ensure the retriable property to all WSs.

Replication of WSs is an alternative strategy to implement forward recovery regardless transactional properties. This technique implies that several equivalent WSs are simultaneously invoked and the response is taken from the first successfully finished one [28,1]. In the modern era of SOA, the cost of developing multiple service versions is greatly reduced. In consequence, equivalent WSs designed/developed independently by different organizations can be readily employed as redundant alternative components for building diversity-based fault tolerant systems. In this context, a replica represents a functionally equivalent WS. Replicas can be used for replication or substitution. Byzantine faults can be also supported by repli-

cating the WS invocations and by ensuring all replicas reach an agreement on the input despite Byzantine faulty replicas. Such an agreement is often referred to as Byzantine agreement.

These strategies can represent advantages or disadvantages depending on user requirements (functionality and *QoS* criteria), nature of services, and execution environment. Thus, they can affect differently the final execution result. For example, for some users partial results may have sense, then, partial backward recovery in the part of the CWS affected by the failure should be the best strategy; some users need the results no matter other *QoS* criteria; hence, forward recovery strategies are appropriated in these cases. The decision of which forward recovery mechanism (retry, substitution, replication) is the most appropriated depends on the execution context, since for other users the total execution time is the most important *QoS*; then depending on the context execution, backward or forward recovery can apply. The unpredictable characteristics of WS environments provide a challenge for optimal fault tolerance strategy determination. Depending on the scenario in which the failure occurred, some recovery techniques are more practical, effective, and efficient than others or some recovery techniques are not possible to apply. Consequently, the execution engine should be capable to adapt fault tolerant decisions for meeting different user requirements and changing environments.

In this paper we study the impact of different recovery strategies on the global *QoS* of CWSs. We focus on CWS execution time; however, other *QoS* criteria can be studied. This study is the first step towards a dynamic recovery decision method.

# 3 CWS Execution Time: An study towards a dynamic recovery decision model

In this section, we present the basis to characterize the CWS and the environment in terms of execution time in order to study the impact of different recovery strategies. This study considers the QoS of component WSs, the state of the execution at the failure moment, and the effect of the recovery strategies on the global QoS of the CWS.

## *3.1 Preliminaries*

QoS criteria describe non-functional WS characteristics (e.g., execution time, price, reliability). The QoS values can vary for a single WS during its lifecycle. Hence, there exist several techniques to keep these values of the QoS parameter as most as possible updated. Particularly, execution time estimation for WSs can be done with analytic, simulation, or test based techniques. In [14], there are proposed methodologies based on a two-factorial analysis and a Gaussian majorization of previous service execution times, enabling the estimation of a WS execution time. In [16], the performance of WSs at different levels (i.e., user level, network level, hardware resource level and software design level) is analyzed. The user level is the one corresponding to the execution time of the WS, which is calculated by

the mean value during a certain period by invoking the actual WS. There is also research in the estimation of the execution time for CWSs. In [25], a time estimation method for CWSs is proposed based on mining historical execution information of the component WSs in the dynamic environment. It takes into account the WS execution time and the network transmission time.

Independently of the technique used for execution time estimation, we assume that each WS is annotated with its estimated execution time, defined as follows.

***Definition 3.1*** **Estimated Execution Time for a WS.** The Estimated Execution Time for a WS, denoted as $WS\_estimated\_TT$, is the execution time value estimated through an estimation technique.

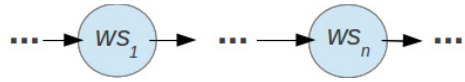
Therefore, the estimated total execution time of a CWS can be calculated in terms of its component WSs and the execution flow depicted by the structure representing the CWS. Recall that in CWSs exist two basic variants of execution scenarios: sequential and parallel. For sequential execution, the estimated execution time is the sum of the estimated execution times of each WS belonging to the sequential path (Figure 3(a) and Equation 1), whilst for parallel execution, the estimated execution time is the maximum estimated execution time of parallel sequential paths (Figure 3(b) and Equation 2) [24].

$$t_{sp} = \sum_{j=1}^{n} t(ws_j) \tag{1}$$

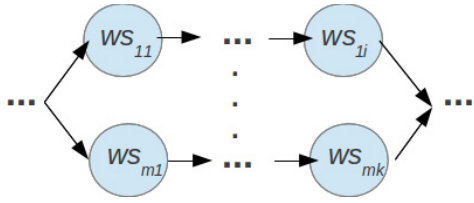
where, $t_{sp}$ is the estimated time of a sequential path with $n$ WSs and $t(ws_j)$ is the estimated execution time of a WS $ws_j$.

$$t_{pp} = \max_{1 \le j \le m} (t_{sp_j}) \tag{2}$$

where, $t_{pp}$ is the estimated time for parallel paths with $m$ sequential paths and $t_{sp_j}$ is the execution time of the sequential path $sp_j$.




(a) Sequential path




(b) Parallel paths

Fig. 3. Sequential and parallel paths

Hence, the total estimated execution time of a CWS is bounded by its sequential path with the maximum cost in terms of estimated execution times of its component WSs. Note that real execution time can be also calculated with equations similar to Eq. 1 and Eq. 2.

### *3.2 Definitions*

We present in this section some definitions in order to formalize our considerations regarding to the characterization of our study.

***Definition 3.2*** **Initial node and final node of a CWS.** Let $G = (V, E)$ be a CWS; the initial and final nodes, denoted as $n_i$ and $n_f$ respectively, are dummy nodes added to a CWS, such that:

- V $= \{n_i, n_f\} \cup V$;
- $\forall\, ws_i \in V : I(ws_i) = \emptyset$; $E = E \cup (n_i, ws_i)$; $n_i$ is the predecessor node to all entry nodes (Def. 2.1) of the CWS;
- $\forall\, ws_i \in V : O(ws_i) = \emptyset$; $E = E \cup (ws_i, n_f)$; $n_f$ is the successor node to all output nodes (Def. 2.1) of the CWS;
- $I(n_i) = \emptyset$; $O(n_i) = \mid ws_i \in V : I(ws_i) = \emptyset \mid$;
  $I(n_f) = \mid ws_i \in V : O(ws_i) = \emptyset \mid$; $O(n_f) = \emptyset$.

Initial and final nodes have only control responsibilities and they are used only to define the start and the end of the CWS execution.

***Definition 3.3*** **Maximum Cost Path of a CWS.** Maximum Cost Path of a CWS is the sequential path from $n_i$ to $n_f$ which has the maximum execution cost.

Note that, there can be more than one maximum cost path in a single CWS.

***Definition 3.4*** **Estimated Execution Time of a CWS.** The Estimated Execution Time of a CWS, denoted as $CWS\_estimated\_TT$, is the sum of the estimated execution time of each WS belonging to the Maximum Cost Path. Its value is calculated according Equation 2.

***Definition 3.5*** **Delta of Estimated Total Execution Time of a CWS.** The Delta of Estimated Total Execution Time for a CWS, denoted as $\Delta\_CWS_{ETET}$, represents the maximum time allowed to exceed for the execution of a CWS, and it is expressed as a percentage of the Estimated Execution Time of the CWS.

$\Delta\_CWS_{ETET}$ denotes the degree of fault tolerated for a specific CWS in terms of execution time. This value can be given by the user or it can be set by the system.

***Definition 3.6*** **Tolerated Execution Time of a CWS.** Let $cws_i$ be a CWS, $CWS\_estimated\_TT_i$ its estimated execution time, and $\Delta\_CWS_{ETET_i}$ its delta of estimated total execution time; the Tolerated Execution Time of $cws_i$, denoted as $CWS\_tolerated\_TT_i$, is defined as:

$$CWS\_tolerated\_TT_i = CWS\_estimated\_TT_i + \Delta\_CWS_{ETET_i}$$

***Definition 3.7*** **Elapsed Real Execution Time for a WS.** Let $ws_i$ be a component WS in a CWS, the Elapsed Real Execution Time for $ws_i$, denoted as $WS\_elapsed\_time_i$, refers to the real invested time since the CWS starts its execution, from $n_i$, until $ws_i$ is invoked.

With the $WS\_elapsed\_time$, it is possible to compute the variation between the estimated execution time and the real execution time taken from the beginning of the execution of a CWS until the actual invocation of each component WS. Note that the $WS\_elapsed\_time_i$ can refer to the real execution time elapsed until the moment in which $ws_i$ was invoked or until the moment the WS failed and an action has to be taken in order to provide fault-tolerance.

***Definition 3.8*** **Estimated Execution Time Left for a WS.** Let $ws_i$ be a component WS in a CWS, the Estimated Execution Time Left for $ws_i$, denoted as $WS\_left\_time_i$, is the maximum cost among the costs of the sequential paths to which $ws_i$ belongs to, measure from the invocation of $ws_i$ until the final node $n_f$ of the CWS.

$WS\_left\_time$ allows to look ahead and calculate *how far* (in terms of execution time) is the end of the execution of the CWS, with respect to each component WS. Note that, depending on the sequential path in which WSs are, this measure could be different for different component WSs. In a specific moment in the line of the execution time, one WS can be near to $n_f$, while other WS can be in the maximum cost path and far from $n_f$.

***Definition 3.9*** **Real Executed Time for a WS.** Let $ws_i$ be a component WS in a CWS, the Real Executed Time for $ws_i$, denoted as $WS\_executed\_time_i$, refers to the real invested time since the $ws_i$ was invoked until $ws_i$ finished successfully or unsuccessfully.

$WS\_executed\_time$ represents the actual execution time invested in the execution of a component WS. If the component WS finishes successfully, its $WS\_executed\_time$ will be the real execution time. In contrast, if the component WS fails, it will represent the time from its invocation until the time when the failure happened.

***Definition 3.10*** **Delta of Estimated Total Execution Time for a WS.** Let be (i) $cws_j$ a CWS, (ii) $CWS\_tolerated\_TT_j$ the tolerated execution time of $cws_j$ (see Def. 3.6), (iii) $ws_i$ a component WS of $cws_j$, (iv) $WS\_elapsed\_time_i$ the elapsed real execution time for $ws_i$ (see Def. 3.7), (v) $WS\_left\_time_i$ the estimated execution time Left for $ws_i$ (see Def. 3.8), and (vi) $WS\_executed\_time_i$ the real executed time for $ws_i$ (see Def. 3.9); the Delta of Estimated Total Execution Time for $ws_i$, denoted as $\Delta\_WS_{ETET_i}$, represents the maximum time allowed to exceed the execution of $ws_i$, in order to do not overcome the $CWS\_tolerated\_TT_j$, and it is expressed as:

$$\Delta\_WS_{ETET_i} = CWS\_tolerated\_TT_i - (WS\_elapsed\_time_i + WS\_left\_time_i + WS\_executed\_time_i) \quad (3)$$

With the $\Delta\_WS_{ETET}$ the execution engine can decide at the moment when a component WS fails, if it is convenient to perform any of the forward recovery strategies without overcoming the tolerated execution time of the corresponding CWS. In this sense, if a WS, let say $ws_i$, fails and $\Delta\_WS_{ETET_i} < 0$ (from Equation 3), it means that performing forward recovery will violate the restriction regarding the tolerated execution time, then a backward recovery has to be performed.

### *3.3 Discussion*

Since we consider a distributed execution engine and the execution control detached from WS implementation, we can suppose that the information needed to decide the appropriated recovery strategy is known by the execution engine at any moment for each component WS. The execution engine is composed by independent software components taking care of each component WS in a CWS. They communicate with each other according to the execution flow depicted by the Composite Web Service Graph to send input parameters or control signals (see Def. 2.1). Thus, along with input parameters or control signals, components of the execution engine can send information needed by the model proposed in Equation 3. The basic metric in our model is the WS execution time. However, WS performance can be influenced by the communication links. We plan to incorporate the metric of data transfer in future work to calculate our metrics more accurately, which in consequence, will support better decisions in case of failures.

By using this first version of our model, the selection of the recovery strategy only considers execution time as the QoS criteria to conserve for a CWS. However, it could be easily extended to consider user requirements/constraints of other QoS criteria (e.g., price, availability) and use other context-information (e.g., data transfer time, probability of faults for the remainder WSs, system reliability and confiability). This kind of information should be useful to help in making better decisions regarding the best recovery strategy choice. Thus, the model can be adaptable and automatically configurable according to QoS user requirements and execution context information.

Additionally, even if this model is designed to respond to silent faults, it could be used to ensure the QoS at the execution time of the CWSs, not only in presence of failures. If the execution conditions are tested before/after the execution of each component WS, a decision for backward recovery could be made if QoS will be violated. Imagine a part of the CWS execution takes much more time that the estimated in a way the total estimated execution time is overcame, in which case the execution has to be halted (a compensantion could be necessary).

As a first step towards a model to provide dynamic decision for fault tolerant strategies, our proposed model allows to confirm that under different environment conditions and different execution scenarios it is possible to adapt the best choice of the recovery strategy when a failure occurs, and still guarantying different user QoS requirements. Next section presents our experimental study.

# 4 Experimental Study

We developed an execution engine that uses our model to support failures during the execution of a CWS. We use Java 6 and the MPJ Express 0.38 library to allow its execution in distributed memory environments. We deployed the execution engine in a cluster of PCs, where the execution control of each WS is executed in a different node of the cluster. All PCs have the same configuration: Intel Pentium 3.4GHz CPU, 1GB RAM, Debian GNU/Linux 6.0, and Java 6. They are connected through a 100Mbps Ethernet interface.

We automatically generated 10 CWSs by using the composition process presented in [6], from synthetic datasets comprised by 800 WSs with 7 replicas each one, for a total of 6400 WSs. These 10 CWSs have 9 or 10 component WSs. All WSs, including replicas, have different QoS values. In particular for experiments, we consider the following QoS parameters:

- Estimated execution time (used in our model as $WS\_estimated\_TT$, see Def. 3.1);
- Availability ($WS\_availability$) representing the probability of the successful execution of a WS (used to simulate different environment conditions);
- Time reliability ($WS\_reliability$) denoting the degree of being capable of maintaining the promised Estimated Execution Time (used to simulate different environment conditions);

Our execution engine simulates different execution environments according those QoS parameters.

We define two conditions: homogeneous environments and heterogeneous environments. In homogeneous environments all WSs have the same availability ($WS\_availability$) and time reliability ($WS\_reliability$); while in heterogeneous environments, $WS\_availability$ and $WS\_reliability$ of all WSs vary. In both cases, the estimated execution time ($WS\_estimated\_TT$) for all WSs varies according an uniform distribution.

We define four scenarios for each kind of environment. Table 1 describes the eight scenarios. Column one enumerates each scenario, column two shows homogeneity level, denoted as $hl$ ($hl = 1$ meaning homogeneous environment and $hl = 0$ means heterogeneous environment). $WS\_availability$ and $WS\_reliability$ for heterogeneous scenarios are shown as the rank of possible values taken. On each scenario, the 10 CWSs were executed 1000 times; it means, 10000 executions for each scenario and for all scenarios 80000 executions.

On each execution of a CWS, none, one, or more than one component WSs could fail according their $WS\_availability$ and $WS\_reliability$. Note that these QoS parameters are considered by our simulator to produce failures, they are not considered in our model, which only uses $WS\_estimated\_TT$ to make recovery decisions.

We design two set of experiments: (i) the first set of experiments were conducted considering only the backward recovery and forward recovery strategies and, (ii) the second set of experiments additionally considers the replication strategy.

To illustrate how the execution engine works based on our model, we will detail

| Scenario | $hl$ | $WS\_availability$ | $WS\_reliability$ |
|---|---|---|---|
| **1** | 1 | 0.8 | 0.9 |
| **2** | 1 | 0.8 | 0.1 |
| **3** | 1 | 0.95 | 0.9 |
| **4** | 1 | 0.95 | 0.1 |
| **5** | 0 | [0.64 - 0.96] | [0.72 - 1.0] |
| **6** | 0 | [0.64 - 0.96] | [0.08 - 0.12] |
| **7** | 0 | [0.76 - 1.0] | [0.72 - 1.0] |
| **8** | 0 | [0.76 - 1.0] | [0.08 - 0.12] |

Table 1
Execution Scenarios

all the steps in the execution process with one of the 10 CWSs; however, similar results and conclusions hold for all other generated CWSs. Figure 4 shows the selected CWS as example. Table 2 shows the $WS\_estimated\_TT$ (in seconds) for each component WS of our selected CWS. Its Maximum Cost Path is shown in Figure 4 by the red component WSs ($ws_3$, $ws_9$, and $ws_{10}$). We extracted the Maximum Cost Path of the CWS example in Figure 5. In this case, the estimated execution time of the CWS example is bounded by its Maximum Cost Path and is calculated as in Def. 3.6:

$$CWS\_estimated\_TT = WS\_estimated\_TT_{ws_3} + WS\_estimated\_TT_{ws_9} + WS\_estimated\_TT_{ws_{10}}.$$

$$CWS\_estimated\_TT = 34980 \text{ secs} + 29650 \text{ secs} + 2472 \text{ secs} = 67102 \text{ secs}.$$

Note that, the failure of $ws_3$, $ws_9$, or $ws_{10}$ will affect the estimated execution time of the whole CWS, more than the failure of any other of its component WSs.

We assume $\Delta\_CWS_{ETET} = 0.1 * CWS\_estimated\_TT$, which means it is inadmissible for an execution of a CWS to take more than 10% of its estimated execution time ($CWS\_estimated\_TT$). Hence, for the CWS example, the tolerated execution time is calculated as in Def. 3.6:

$$CWS\_tolerated\_TT = CWS\_estimated\_TT + \Delta\_CWS_{ETET} = 67102 \text{ secs} + 6710{,}2 \text{ secs} = 73812{,}2 \text{ secs}$$

Let us explain the first set of experiments with the CWS example.

**Example 1:**

*Suppose $ws_9$ fails when it had executed 18000 secs. Thus, we have:*

- *$WS\_elapsed\_time_{ws_9}$ = 34980 secs (ws_3 execution time, see Def. 3.7);*
- *$WS\_left\_time_{ws_9}$ = 29650 secs (its own execution time) + 2472 secs ($ws_{10}$ execution time) = 32122 secs (see Def. 3.8); and*
- *$WS\_executed\_time_{ws_9}$ = 18000 secs (see Def. 3.9).*

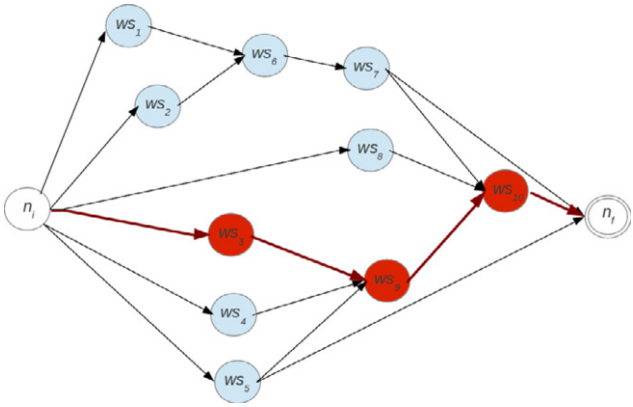



Fig. 4. Illustrative CWS: This CWS is one of the 10 automatically generated

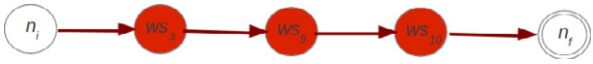



Fig. 5. Maximum Cost Path of CWS in Figure 4

| component WS | $WS\_estimated\_TT$ (secs) |
|---|---|
| $ws_1$ | 8080 |
| $ws_2$ | 8020 |
| $ws_3$ | 34980 |
| $ws_4$ | 7570 |
| $ws_5$ | 12990 |
| $ws_6$ | 836 |
| $ws_7$ | 1388 |
| $ws_8$ | 13330 |
| $ws_9$ | 29650 |
| $ws_{10}$ | 2472 |

Table 2
WSs estimated execution time

*As we showed before* $CWS\_tolerated\_TT = 73812,2$ *secs, then from Equation 3, we have:*

$$\begin{aligned}\Delta\_WS_{ETET_{ws_9}} &= 73812,2 - (34980 + 32122 + 18000) \\ &= -11289,8 secs\end{aligned}$$

*Because* $\Delta\_WS_{ETET_{ws_9}} < 0$*, the execution engine decides backward recovery. If forward recovery is applied by retrying the execution of* $ws_9$ *or selecting a substitute that in the best case has the same estimated execution time of* $ws_9$*, the expected total execution time for the CWS will be exceeded.*

**Example 2:**

*Now suppose that the failure was for $ws_{10}$ when it was almost finishing its execution, at 2200 secs. Thus, we have:*

- $WS\_elapsed\_time_{ws_{10}}$ = *64630 secs ($ws_3$ + $ws_9$ execution times);*
- $WS\_left\_time_{ws_{10}}$ = *2472 secs (only its own execution time because the successor of $ws_{10}$ is $n_f$); and*
- $WS\_executed\_time_{ws_{10}}$ = *2200 secs.*

*As we explain before $CWS\_tolerated\_TT = 73812,2$ secs. Finally, from Equation 3, we have:*

$$\begin{aligned}\Delta\_WS_{ETET_{ws_9}} &= 73812,2 - (64630 + 2472 + 2200)\\ &= 4510,2 secs.\end{aligned}$$

*In this case $\Delta\_WS_{ETET_{ws_{10}}} > 0$, then the execution engine decides forward recovery, by retrying the execution of $ws_{10}$ or selecting a substitute, because the expected total execution time for CWS will be kept. When a substitute is selected, its estimated execution time is the one considered in our model (Equation 3).*

Note that in these examples, the faulty component WSs belong to the Maximum Cost Path; however, the same procedure works for any component WS.

For experimental intentions, when backward recovery was decided, the execution engine forced a forward recovery to show how much will be impacted the total execution time of the CWS if backward recovery is not performed.

Table 3 shows the percentage of variation of CWSs Total Execution Time regarding its estimated time for the first set of experiments in all scenarios as the average of the executions of the 10 CWSs. Table 3 presents in:

- Column one, each scenario;
- Column two, the results related to executions without faults ($NoFaults$).
- Column three, the results related to executions in which forward recovery was decided ($Forward$);
- Column four, the results related to executions in which backward recovery was decided but the execution engine forced a forward recovery ($Forward$), showing the impact in the total execution time if backward recovery is not performed;
- Column five, the percentage of executions where the time constraint was violated ($TimeExceeded$)

Columns two ($NoFaults$), three($Forward$), and four ($Forward$) represent the variation for the CWS estimated execution time regarding its estimated time; for example, the value of column two line one indicates that the real execution time was 2.6% more of the estimated execution time.

These results show that there is little variation in the total execution time between executions without failures ($NoFaults$) and executions in which forward recovery was decided ($Forward$). These values are relatively small, which means that with the forward recovery strategy, performing re-execution or substitution of WSs did not have a negative impact on the whole execution time regarding

| Scn | *NoFaults* | *Forward* | *Backward* | *TimeExceeded* |
|---|---|---|---|---|
| **1** | 2.6 | 2.8 | 42.45 | 48 |
| **2** | 3.5 | 3.5 | 48.80 | 52 |
| **3** | 2.4 | 2.9 | 40.98 | 14 |
| **4** | 3.4 | 3.9 | 41.87 | 15 |
| **5** | 3.0 | 3.0 | 45.92 | 46 |
| **6** | 3.7 | 3.6 | 47.18 | 44 |
| **7** | 3.86 | 3.91 | 38.24 | 15 |
| **8** | 3.1 | 3.43 | 38.88 | 18 |

Table 3
Percentage of variation of CWSs Total Execution Time with Backward and Forward Recovery

the time constraint. Total execution time varied at maximum in 3.91% from the $CWS\_estimated\_TT$ with an expected $\Delta\_CWS_{ETET} = 10\% * CWS\_estimated\_TT$. In contrast, $CWS\_estimated\_TT$ was exceeded in more than 38%, when the model suggested backward recovery but the system forced a forward recovery (*Backward*), exceeding the time constraint in at least 14% of the times (*TimeExceeded*).

The second set of experiments additionally takes into account replication. For component WSs in the Maximum Cost Path, at the moment of its invocation, the execution engine evaluates with the model (as we illustrated in Example 1) "what will happen if this WS fails"; if its failure will cause a backward recovery, then the execution engine invokes simultaneously its replicas. It means, replicas of the WSs in the Maximum Cost Path can be executed in parallel along with their corresponding original WS, as shown in Figure 6. Only the answer of the first WS that finishes correctly is taken into account to continue with the execution of the CWS. For any other component WS, not belonging to the Maximum Cost Path, the replication decision is evaluated if it fails. If forward recovery keeps the total execution time constraint, no replica is invoked.

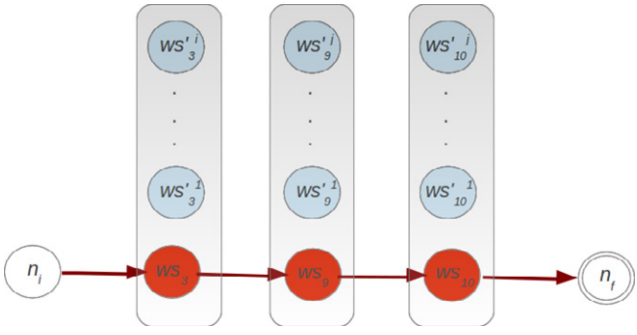



Fig. 6. Replication of WSs in Maximum Cost Path

Tables 4, 5, and 6 show the results allowing the replication strategy to be

performed using one replica, three replicas, and seven replicas per WS, respectively. The columns *Scn*, *NoFault*, and *Forward* have the same meaning as in Table 3. The column *BackwardReplicat* represents the percentage of the total execution time using replication to go forward in the cases the model suggested backward recovery because the time constraint was going to be violated. The column *Replicat* shows percentage of the total execution time using replication. The column *AllFail* represents the percentage of executions where the replication strategy could not satisfy the time constraint because all replicas of one WS failed. The column *ReplicatNeed* expresses the percentage of the total number of executions where the result of a replica was used instead of the original WS, because it failed.

As in the first set of experiments, the forward recovery strategy (column *Forward*, in the three tables) did not produce a high impact on the total execution time, the worst case was 3.97% with seven replicas, while the tolerated time is 10%. Once again, when the model detected that the backward recovery should be executed and the system forced forward recovery, the *CWS_estimated_TT* was exceeded in more than 30% when 1 or 3 replicas were used (column *BackwardReplicat* in Tables 4 and 5); means that the replication strategy did not help to satisfy the time constraint because at least one of the critical WSs and all its replicas failed in at most 15% of the times (column *AllFail* in Tables 4 and 5); with seven replicas, it never happened that backward recovery was needed (column *BackwardReplicat* in Tables 6) because the original WS and all its replicas failed (column *AllFail* in Table 6). When the original WS failed and one of its replicas finished successfully, the total execution time is not impacted a lot (column *Replicat* in all tables), the worst case was 3.81% of variation from the estimated total execution time and this happened at least 30% times from the cases in which all replicas where invoked along with the original WS (column *ReplicatNeed* in all tables); this result shows that invoking replicas in normal conditions can prevent faults without almost any impact in the total execution time.

Figure 7 illustrates the percentage of executions where the time constraint was violated under the eight scenarios without using replication (first set of experiments) and under the same scenarios enabling replication with one, three, and seven replicas per WS (second set of experiments). Remember that in all cases, the model suggested backward recovery but the system forced to forward recovery. As explained before, the replication strategy is chosen as prevention, if the WS to be executed belongs to the Maximum Cost Path; therefore, there is more probability of a successful execution of replicated WSs and then, been able to fulfill the time constraint.

The difference between the bars in Figure 7 highlights the improvement achieved by adding the replication strategy to the set of possible actions to perform in order to maintain the time constraint. Table 7 shows the approximate failure probabilities for a single node using different numbers of replicas and assuming that all original WSs have the worst failure probabilities considered for these experiments, which is 0.2. This is the reason behind the considerable decrease on the time constraint violation percentages showed in Figure 7, specially for executions with 3 and 7

| Scn | *No Fault* | *Forward* | *Backward Replicat* | *Replicat* | *All Fail* | *Replicat Need* |
|---|---|---|---|---|---|---|
| **1** | 2.77 | 2.81 | 37.82 | 3.10 | 12 | 42 |
| **2** | 3.53 | 3.62 | 38.15 | 3.81 | 8 | 38 |
| **3** | 2.62 | 2.63 | 30.65 | 2.91 | 1 | 33 |
| **4** | 3.49 | 3.52 | 33.49 | 3.54 | 4 | 33 |
| **5** | 2.62 | 2.63 | 36.21 | 2.69 | 13 | 35 |
| **6** | 3.44 | 3.71 | 38.7 | 3.44 | 15 | 35 |
| **7** | 2.63 | 2.63 | 38.85 | 2.68 | 2 | 38 |
| **8** | 3.39 | 3.44 | 37.24 | 3.45 | 1 | 34 |

Table 4
Execution with Replication (1 replicas)

| Scn | *No Fault* | *Forward* | *Backward Replicat* | *Replicat* | *All Fail* | *Replicat Need* |
|---|---|---|---|---|---|---|
| **1** | 2.95 | 2.92 | $\emptyset$ | 3.03 | 0 | 34 |
| **2** | 3.68 | 3.72 | 43.54 | 3.10 | 1 | 33 |
| **3** | 2.93 | 2.94 | $\emptyset$ | 2.91 | 0 | 37 |
| **4** | 3.69 | 3.68 | $\emptyset$ | 3.69 | 0 | 32 |
| **5** | 2.75 | 2.80 | 42.11 | 3.11 | 1 | 33 |
| **6** | 3.44 | 3.44 | $\emptyset$ | 3.50 | 0 | 34 |
| **7** | 2.63 | 2.65 | $\emptyset$ | 2.71 | 0 | 35 |
| **8** | 3.39 | 3.44 | $\emptyset$ | 3.45 | 0 | 30 |

Table 5
Execution with Replication (3 replicas)

replicas.

All these experiments illustrate that if a wrong recovery strategy is executed, the impact in the global QoS parameters of a CWS can be highly affected. They also show how suitable is to know information about the execution environment to apply prevention strategies (as replication) that will not affect the QoS parameters in neither in normal nor in faulty conditions. Thus, a model that can take into account context-information will help the execution engine to make decision regarding the best recovery strategy, while global QoS parameters are maintained.

| Scn | *No Fault* | *Forward* | *Backward Replicat* | *Replicat* | *All Fail* | *Replicat Need* |
|---|---|---|---|---|---|---|
| **1** | 2.95 | 2.92 | ∅ | 3.03 | 0 | 39 |
| **2** | 3.68 | 3.72 | ∅ | 3.10 | 0 | 38 |
| **3** | 2.93 | 2.94 | ∅ | 2.91 | 0 | 33 |
| **4** | 3.69 | 3.68 | ∅ | 3.69 | 0 | 34 |
| **5** | 2.75 | 2.80 | ∅ | 3.11 | 0 | 39 |
| **6** | 4.00 | 3.97 | ∅ | 3.77 | 0 | 37 |
| **7** | 2.63 | 2.61 | ∅ | 2.71 | 0 | 33 |
| **8** | 3.39 | 3.44 | ∅ | 3.45 | 0 | 36 |

Table 6
Execution with Replication (7 replicas)

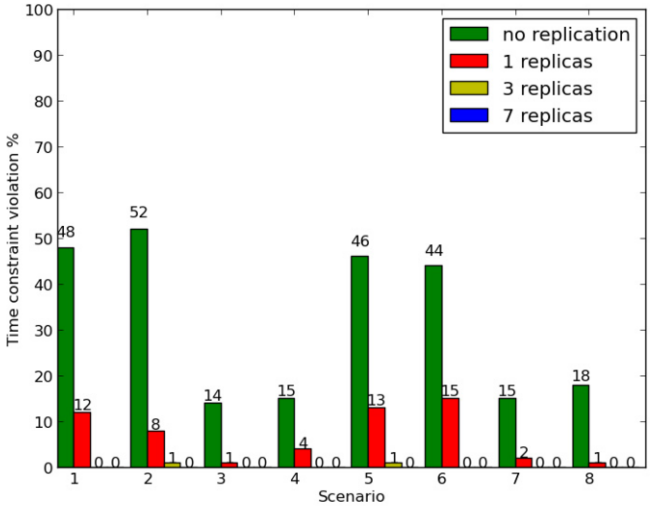



Fig. 7. Time constraint violation without and with replication

Note that in our experiments, results were not impacted by the homogeneity level. Similar results and conclusions were obtained from both cases. This experience can represent another benefit to mention of our model: it is independent from how heterogenous is the execution environment. We plan to test our model with more exhaustive cases to probe these preliminary suppositions.

| $replicas$ | $failureprobability$ |
|---|---|
| 0 | ≈0.2 |
| 1 | ≈0.04 |
| 2 | ≈0.008 |
| 3 | ≈0.008 |
| 4 | ≈0.0016 |
| 5 | ≈0.00002 |
| 6 | ≈0.00002 |
| 7 | ≈0.00002 |

Table 7
Node failure probabilities

## 5 Related Work

There exist many works related to fault tolerance execution for CWSs that implement a combination of several recovery strategies.

FENECIA framework [13] provides an approach for managing fault tolerance and QoS in the specification and execution of CWSs. FENECIA introduces WS-SAGAS, a transaction model based on arbitrary nesting, state, vitality degree, and compensation concepts to specify fault tolerant CWS as a hierarchy of recursively nested transactions. To ensure a correct execution order, the execution control of the resulting CWS is hierarchically delegated to distributed engines that communicate in a peer-to-peer fashion. A correct execution order is guaranteed in FENECIA by keeping track of the execution progress of a CWS and by enforcing forward and backward recovery. To manage failures during the runtime it allows the execution retrial with alternative candidates. FACTS [15] is another framework for fault tolerant composition of transactional WSs based on FENECIA transactional model. It combines exception handling strategies and a service transfer based termination protocol. When a fault occurs at run-time, it first employs appropriate exception handling strategies to repair it. If the fault cannot be fixed, it brings the CWS back to a consistent termination state according to the termination protocol (by considering alternative services, replacements, and compensation).

In [4], a compensation workflow is built. This workflow has the lowest compensation cost, which includes the cost of failed WS and the total cost of compensating the previously executed WSs. There exist some recent works related to the compensation mechanism of CWSs based on Petri-Net formalism [17,19,22]. The compensation process is represented by Paired Petri-Nets demanding that all component WSs have to be compensable.

There exist some works that implement different fault tolerant strategies based on WS-BPEL technologies and consider highly dynamic environments as cloud com-

puting. In [12], WS invocations are intercepted by an integrated software component to the BPEL engine. If a failure occurs during an invocation, it is handled by this extension according to policies that take into account specific characteristics of the cloud environment. This solution is not transparent, it is strongly attached to the specific BPEL engine implementation. In [1], a replication strategy is used and a rollback workflow is automatically created considering the service dependencies. An actively replicated platform is presented, in which all replicas of a WS are simultaneously invoked. Only results of the first replica finished are accepted, other executions are halted or ignored. The process of replication and coordination of replicas is implemented transparently to users and independent to WS implementation.

All those previously described works do not consider the dynamism of the execution context environment to adapt the decision regarding to which recovery strategy is the most appropriated. They implement specific and static fault tolerance strategies.

In [27], it is defined an adaptive and dynamic fault tolerance strategy based on execution time, failures probability, and resource consumption parameters. Users specify weights that represent their requirements over those three parameters. An additive weighting function of the parameters. This work presents an experimental study to show the feasibility of determining recovery strategies that comply user needs. This work is the most related to our goals; however, it is meant for the fault tolerance of single WSs, not for the fault tolerance of entire CWSs, as our study does.

## 6 Conclusions and Future Work

In this work, we have presented a preliminary model to dynamically decide which recovery strategy is the most appropriated according to execution time restrictions. We conduced an experimental study towards the definition of a more complex and complete model to adapt the fault-tolerance strategy to context-information. This automatic decision takes into account the impact of the recovery strategy on QoS parameters and user preferences or system constraints, like the time constraint presented in this analysis. The alternative recovery strategies considered in our study were backward and forward recovery based on transactional properties and replication to support forward recovery. Our experimental study demonstrates that a model that can take into account context-information will help the execution engine to make decision regarding the best recovery strategy, while global QoS parameters are maintained. They also show how suitable is to know information about the execution environment to apply prevention strategies (as replication) that will affect the QoS parameters in neither normal nor faulty conditions.

We plan to test our model with more exhaustive test cases to prove these preliminary suppositions and to extend it to consider other context information and other QoS requirements, as the availability and reliability of component WSs, in order to better support the decision making. We also plan to adapt the model to integrate other recovery strategies such as checkpointing techniques and tolerate other kind

of failures such as Byzantine faults. Finally, we plan to perform experiments using real data and uses cases.

# References


[1] Behl, J., T. Distler, F. Heisig, R. Kapitza and M. Schunter, *Providing fault-tolerant execution of web-service–based workflows within clouds*, in: *Proc. of the 2nd Int. Workshop on Cloud Computing Platforms (CloudCP)*, 2012.

[2] Benjamins, R., J. D. E. Dorner, J. Domingue, D. Fensel, O. López, R. Volz, A. Wahler and M. Zaremba, *Service web 3.0*, Technical report, Semantic Technology Institutes International (2007).

[3] Blanco, E., Y. Cardinale and M.-E. Vidal, "Aggregating Functional and Non-Functional Properties to Identify Service Compositions," IGI BOOK, 2011 pp. 1–36.

[4] Bushehrian, O., S. Zare and N. K. Rad, *A workflow-based failure recovery in web services composition*, Journal of Software Engineering and Applications **5** (2012), pp. 89–95.

[5] Cardinale, Y., J. El Haddad, M. Manouvrier and M. Rukoz, *Web service selection for transactional composition*, Elsevier Science-Procedia Computer Science Series (preliminary version presented in Int. Conf. on Computational Science(ICCS) **1** (2010), pp. 2689–2698.

[6] Cardinale, Y., J. El Haddad, M. Manouvrier and M. Rukoz, *CPN-TWS: A colored petri-net approach for transactional-qos driven web service composition*, Int. Journal of Web and Grid Services **7** (2011), pp. 91–115.

[7] Cardinale, Y., J. El Haddad, M. Manouvrier and M. Rukoz, "Transactional-aware Web Service Composition: A Survey," IGI Global - Advances in Knowledge Management (AKM) Book Series, 2011 pp. 2–20.

[8] Cardinale, Y. and M. Rukoz, *Fault tolerant execution of transactional composite web services: An approach*, in: *Proc. of The Fifth Int. Conf. on Mobile Ubiquitous Computing, Systems, Services qnd Technologies (UBICOMM)*, 2011.

[9] Cardinale, Y. and M. Rukoz, *A framework for reliable execution of transactional composite web services*, in: *MEDES*, 2011, pp. 129–136.

[10] Cardinale, Y. and M. Rukoz, *A framework for reliable execution of transactional composite web services*, in: *Proc. of The Int. ACM Conf. on Management of Emergent Digital EcoSystems (MEDES)*, 2011.

[11] El Haddad, J., M. Manouvrier and M. Rukoz, *TQoS: Transactional and QoS-aware selection algorithm for automatic Web service composition*, IEEE Trans. on Services Computing **3** (2010), pp. 73–85.

[12] Juhnke, E., T. Dörnemann and B. Freisleben, *Fault-tolerant bpel workflow execution via cloud-aware recovery policies*, in: *EUROMICRO-SEAA* (2009), pp. 31–38.

[13] Lakhal, N. B., T. Kobayashi and H. Yokota, *FENECIA: failure endurable nested-transaction based execution of compo site Web services with incorporated state analysis*, VLDB Journal **18** (2009), pp. 1–56.

[14] Lelli, F., G. Maron and S. Orlando, *Towards response time estimation in web services*, in: *Web Services, 2007. ICWS 2007. IEEE International Conference on*, 2007, pp. 1138–1139.

[15] Liu, A., Q. Li, L. Huang and M. Xiao, *FACTS: A Framework for Fault Tolerant Composition of Transactional Web Services*, IEEE Trans. on Services Computing **3** (2010), pp. 46–59.

[16] Mangal1, D. and R.P.Mahajan, *A novel approach for performance estimation of soap-based web services*, in: *International Journal of Emerging Technology and Advanced Engineering, Volume 2, Issue 2*, 2012, pp. 1138–1139.

[17] Mei, X., A. Jiang, S. Li, C. Huang, X. Zheng and Y. Fan, *A compensation paired net-based refinement method for web services composition*, Advances in Information Sciences and Service Sciences **3** (2011).

[18] OASIS, *Web Services Businnes Process Execution Language (WS-BPEL), Version 2.0*, http://docs.oasis-open.org/wsbepel/2.0/wsbpel-v2.0.html (2007), oASIS Standard.

[19] Rabbi, F., H. Wang and W. MacCaull, *Compensable workflow nets*, in: *Formal Methods and Software Engineering - 12th Int. Conf. on Formal Engineering Methods*, LNCS **6447**, 2010 pp. 122–137.

[20] Schafer, M., P. Dolog and W. Nejdl, *An environment for flexible advanced compensations of web service transactions*, ACM Transactions on the Web **2** (2008).

[21] Sherman, D., *Bpel: Make your services flow. composing web services into business flow*, Journal in Web Services **3** (2003), pp. 16–21.

[22] Wang, Y., Y. Fan and A. Jiang;, *A paired-net based compensation mechanism for verifying Web composition transactions*, in: *The 4th Int. Conf. on New Trends in Information Science and Service Science*, 2010.

[23] Yu, Q., X. Liu, A. Bouguettaya and B. Medjahed, *Deploying and managing web services: issues, solutions, and directions*, The VLDB Journal **17** (2008), pp. 537–572.
URL http://dx.doi.org/10.1007/s00778-006-0020-3

[24] Zeng, L., B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam and H. Chang, *Qos-aware middleware for web services composition*, IEEE Trans. Softw. Eng. **30** (2004), pp. 311–327.
URL http://dx.doi.org/10.1109/TSE.2004.11

[25] Zhang, X., Y. Yin and B. Zhang, *A service mining approach for time estimation of composite web services*, in: J. Zhang, editor, *Applied Informatics and Communication*, Communications in Computer and Information Science **228**, Springer Berlin Heidelberg, 2011 pp. 486–492.

[26] Zhao, Z., J. Wei, L. Lin and X. Ding, *A concurrency control mechanism for composite service supporting user-defined relaxed atomicity*, in: *The 32nd Annual IEEE Int. Computer Software and Applications Conf.*, 2008, pp. 275–278.

[27] Zheng, Z. and M. R. Lyu, *An adaptive qos-aware fault tolerance strategy for web services*, Empirical Softw. Engg. **15** (2010), pp. 323–345.
URL http://dx.doi.org/10.1007/s10664-009-9126-8

[28] Zhou, W. and L. Wang, *A byzantine fault tolerant protocol for composite web services*, in: *International Conference on Computational Intelligence and Software Engineering (CiSE)*, 2010, pp. 1–4.