World Wide Web (2016) 19:89–109
DOI 10.1007/s11280-015-0329-1

Modeling dynamic recovery strategy for composite web services execution

Rafael Angarita · **Marta Rukoz** · **Yudith Cardinale**

Received: 15 April 2014 / Revised: 20 October 2014 /
Accepted: 6 January 2015 / Published online: 1 February 2015
© Springer Science+Business Media New York 2015

Abstract During the execution of Composite Web Services (CWS), a component Web Service (WS) can fail and can be repaired with strategies such WS retry, substitution, compensation, roll-back, replication, or checkpointing. Each strategy behaves differently on different scenarios, impacting the CWS QoS. We propose a non intrusive dynamic fault tolerant model that analyses several levels of information: environment state, execution state, and QoS criteria, to dynamically decide the best recovery strategy when a failure occurs. We present an experimental study to evaluate the model and determine the impact on QoS parameters of different recovery strategies; and evaluate the intrusiveness of our strategy during the normal execution of CWSs.

Keywords Composite web services · Fault tolerance · Dynamic recovery techniques · QoS monitoring · Adaptative systems · Self-healing systems

1 Introduction

Web Services (WS) and semantic technologies have emerged to create an environment where users and applications can search, compose, and execute services in an automatic and seamless manner. SOA is expected to be a place where many WSs compete to offer a wide

R. Angarita (✉) · M. Rukoz
LAMSADE UMR 7243, Université Paris-Dauphine, Paris, France
e-mail: rafael.angarita@lamsade.dauphine.fr

M. Rukoz
e-mail: marta.rukoz@lamsade.dauphine.fr

M. Rukoz
Université Paris Ouest Nanterre la Défense, Nanterre, France

Y. Cardinale
Departamento de Computación, Universidad Simón Bolívar, Caracas, Venezuela
e-mail: yudith@ldc.usb.ve

range of similar functionalities. Moreover, WSs from distributed locations can be composed to create new value-added Composite WSs (CWS) [5].

During the execution of a CWS, different situations may cause failures on its component WSs. However, a fault tolerant CWS is one that, upon a WS failure, ends up the execution (e.g., by retrying, substituting, or replicating the faulty WS) or it aborts and leaves the system in a safe state (e.g., by rolling back or compensating the faulty and executed WSs) [10]. Sometimes, partial responses may have sense for user queries; hence, checkpointing can be used as an alternative fault tolerance technique [6, 20, 22]. Because WSs can be created and updated on-the-fly, the execution system needs to dynamically detect changes during run-time and adapt the execution to the availability of the existing WSs. The highly dynamic nature of Internet and the compositional nature of WSs make these static fault tolerance strategies unpractical in real-world environments. In this context, recovery strategies have different behavior according to the execution state at the moment of the failure (e.g., how many component WSs have been successfully executed); the environment state (e.g., network load); and the impact of the recovery strategy in the CWS QoS.

Some questions emerge to decide which recovery strategy is the best in terms of the impact on CWS QoS: are all recovery techniques equally practical, effective, and efficient? When is it better to apply backward (or forward) recovery? Is it replication the best strategy? The unpredictable characteristics of SOA environments provide a challenge for optimal fault tolerance strategy determination. It is necessary more general and smarter fault tolerance strategies, which are context-information aware and can be dynamically and automatically reconfigured to meet different user requirements. It is important to define a dynamic fault tolerant strategy which takes into account context-information to accordingly decide the best one.

In previous work, we have presented a preliminary model to analyze execution information when a failure occurs, selecting the best recovery strategy in terms of impact on the CWS QoS [1]. In this paper, we extend our model to incorporate environment state information, more execution state information, and several QoS criteria, to obtain a self-healing model. We present an experimental study to evaluate our model and determine the impact on QoS parameters of different recovery strategies, and to evaluate the performance of doing the necessary computation to make it work. The experimental results show that, under different conditions, recovery strategies behave differently and the model always chooses the best recovery strategy.

2 Fault tolerance for composite web services

2.1 Composite web service

A Composite Web Service (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 with structures such as workflows, graphs, or Petri Nets, indicating, for example, the control flow, data flow, WSs execution order, and WS behavior. The structure representing a CWS can be manually or automatically generated. Users can manually specify *how* the functionality of WSs are combined or a “composer agent” can automatically build a CWS according to a query. The execution of a CWS is carried out by an “execution engine”.

WSs are described according to their functionalities (e.g., input and output attributes, preconditions, effects) and QoS parameters. QoS parameters describe the WS execution quality

in terms of response time, cost, reliability, throughput, trust, etc. In this context, users can demand functional and non-functional requirements. Thus, WSs delivering the same functionality must be managed to ensure efficient implementations of CWSs. We consider a user query expressed as inputs given by the user, outputs desired by the user (functional requirements), and the importance given by the user to QoS criteria (non-functional requirements). We define query and CWS as follows:

Definition 1 Query. A Query Q is a 3-tuple $\left(I_Q, O_Q, W_Q\right)$, where I_Q is the set of input attributes, O_Q is the set of output attributes whose values have to be produced by the system, and $W_Q = \{(w_i, q_i) \mid w_i \in [0,1]$ with $\sum_i w_i = 1$ and q_i is a QoS criterion$\}$ represents weights over QoS criteria.

Definition 2 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 WSs, such that $V = \{ws_i, i = 1..m\}$ and ws_i is a component WS.
- Arcs in E denote the execution flow among $ws_i \in V$. Execution flow is defined by data or control flow relationships between two WSs. Data flow relationship is defined in terms of ws_i 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 the end of execution of another one; control flow can be designated by control signals or control data. Thus, if $ws_i, ws_j \in V$ and $(ws_i, ws_j) \in E$, and $O(ws_i)$ 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 $\exists\, ws_i \in V : I(ws_i) \cap I_Q \neq \emptyset$.
- Output nodes represent WSs that produce the final desired output attributes to the user, then $\exists\, ws_i \in V : O(ws_i) \cap O_Q \neq \emptyset$.

To define the start and the end of a CWS, initial n_i and final n_f nodes are added to the CWS Graph. These nodes have only control responsibilities to manage the start and the end of CWS executions. They are defined as follows:

Definition 3 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 the CWS, such that:

- V = $\{n_i, n_f\} \cup V$;
- $\forall\, ws_i \in V : I(ws_i) \cap I_Q \neq \emptyset; E = E \cup (n_i, ws_i)$. n_i is the predecessor node to all entry nodes (Definition 2) of the CWS;
- $\forall\, ws_i \in V : O(ws_i) \cap O_Q \neq \emptyset; E = E \cup (ws_i, n_f)$. n_f is the successor node to all output nodes (Definition 2) of the CWS;
- $I(n_i) = \emptyset; \mid O(n_i) \mid = \mid ws_i \in V : I(ws_i) \cap I_Q \neq \emptyset \mid$;
  $\mid I\left(n_f\right) \mid = \mid ws_i \in V : O(ws_i) \cap O_Q \neq \emptyset \mid O\left(n_f\right) = \emptyset$.

Workflows, bipartite graphs, and Petri Nets, the most popular structures used to represent CWSs, can be matched to our graph definition, even if the relationship among WSs is defined by data flow and/or control flow.

2.2 CWS execution control

The execution of a CWS implies the invocation of all component WSs according to the execution flow imposed by the CWS Graph (Definition 2). Thus, there exist two basic variants of execution scenarios: sequential and parallel. In a sequential scenario, some WSs cannot be invoked until previous WSs have finished because they work on the results of those previous WSs, or due to restrictions control sequentially imposed. In a parallel scenario, several WSs can be invoked simultaneously because they do not have flow dependencies.

2.3 Failures in CWSs

During the execution of a CWS, failures can occur at hardware, operating system, WSs, execution engine, and network levels. These failures result in reduced performance, and can cause different behavior in the execution.

- **Silent faults** are generic to all WSs. They cause WSs to not respond, because they are not available or a crash occurred in the platform. Some examples are communication timeout, service unavailable, bad gateway, and server error. Silent faults can be identified by the "execution engine".
- **Logic faults** are specific to different WSs, and are caused by errors in inputs attributes (e.g., bad format, out of valid range, calculation faults) and byzantine faults (WSs respond to invocation in a wrong way). Various exceptions thrown by WSs to users are classified as logic-related faults. It is difficult for the "execution engine" to identify such type of faults.

We consider: (i) WSs can suffer silent failures (logic faults are not considered); (ii) the "execution engine", in charge of the CWS execution, runs far from WS hosts in reliable servers such as computer clusters, it does not fail, its data network is highly secure, and it is not affected by WSs faults; (iii) we suppose that the information needed to choose a recovery strategy is known by the "execution engine" at any moment for each WS.

2.4 Fault tolerant CWS execution

The execution control of CWSs can be centralized or distributed. Centralized approaches consider a coordinator managing the whole execution [21, 26]. In distributed approaches, the execution process proceeds with the collaboration of several participants without a central coordinator [4, 7]. On the other hand, the execution control could be attached to WSs [13, 14] or independent of them [9]. Some execution engines are capable of managing failures during the execution. They can be based on exception handling [8, 19], transactional properties [7, 11], or a combination of both approaches [13, 14]. In previous research in the field of fault tolerant CWSs, only low level programming constructs such as exception handling (for example in WSBPEL) were considered. Exception handling is normally explicitly specified at design time, and is normally used to manage logic faults, which are specific to each WS.

More recently, the reliability of CWSs has 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 CWSs have emerged, such as transactional properties and replication.

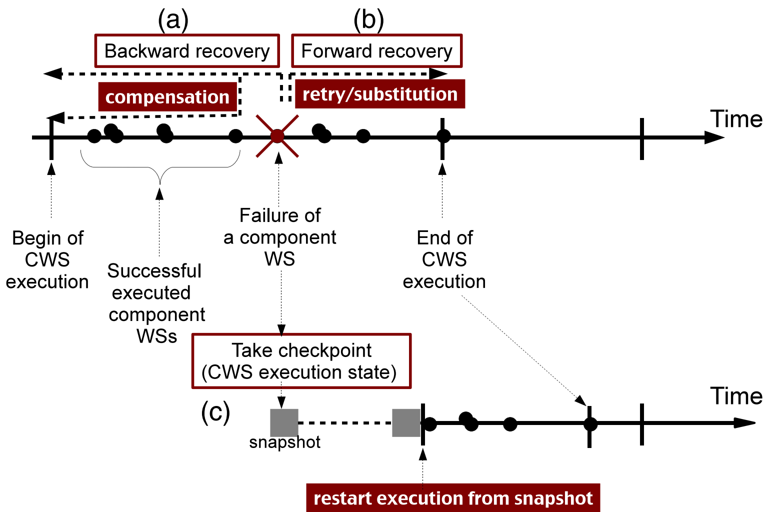

Figure 1 Recovery Techniques

Transactional properties implicitly describe behavior in case of failures, and are used to ensure the classical Atomicity (all-or-nothing[1]) transactional property. When transactional properties are not considered, the system consistence is a responsibility of users/designers. The most used transactional properties for simple WSs are: **pivot**, **compensable**, and **retriable** [11]. A WS is **pivot** (p) if once the WS completes successfully, its effects remain forever and cannot be undone, if it fails, it has no effect at all. A WS is **compensable** (c) if it exists another WS which can undo its successful execution. A WS is **retriable** (r) if it guarantees a successful termination after a finite number of invocations; the **retriable** property can be combined with properties p and c defining **pivot retriable** (pr) and **compensable retriable** (cr) WSs.

WSs that provide transactional properties are useful to guarantee reliable CWS executions, ensuring the whole system consistent state even in presence of failures. An aggregated transactional property is assigned to a CWS in terms of transactional properties and control flow of its WSs [11]. The basic recovery techniques supported by transactional properties of CWSs are backward and forward recovery. A transactional CWS always allows backward recovery by compensating the effects produced by the faulty and successful WSs before the failure (Figure 1a). A transactional CWS allows forward recovery if it is retriable, by retrying the faulty WS (Figure 1b).

All these recovery techniques ensure the all-or-nothing property to keep the system consistency; however, checkpointing techniques can be implemented to relax the all-or-nothing property for transactional CWSs [20] or independent of transactional properties [6, 22], and still provide fault tolerance. In both cases, when a failure occurs, snapshots that contain advanced execution state (including partial results) are taken and returned to the user. The checkpointed CWS can be resumed from an advanced point of execution, without affecting its aggregated transactional property (Figure 1c) .

[1]Each component WS in a CWS must either complete successfully or have no effect whatsoever.

In this work, we consider a distributed "execution engine" composed by independent components taking care of each WS in a CWS. They communicate with each other, according to the execution flow depicted by the CWS Graph, to send input parameters or control signals. The components of the "execution engine" also manage information needed by the fault tolerance mechanisms. The execution control is detached, independent, and transparent to WS implementations (Figure 2).

3 Classification of context information

QoS Criteria QoS values describe non-functional WS characteristics. We consider execution time, price, reputation, and transactional propierties as QoS criteria. They have been calculated before the execution of CWSs, and their values are known at run-time.

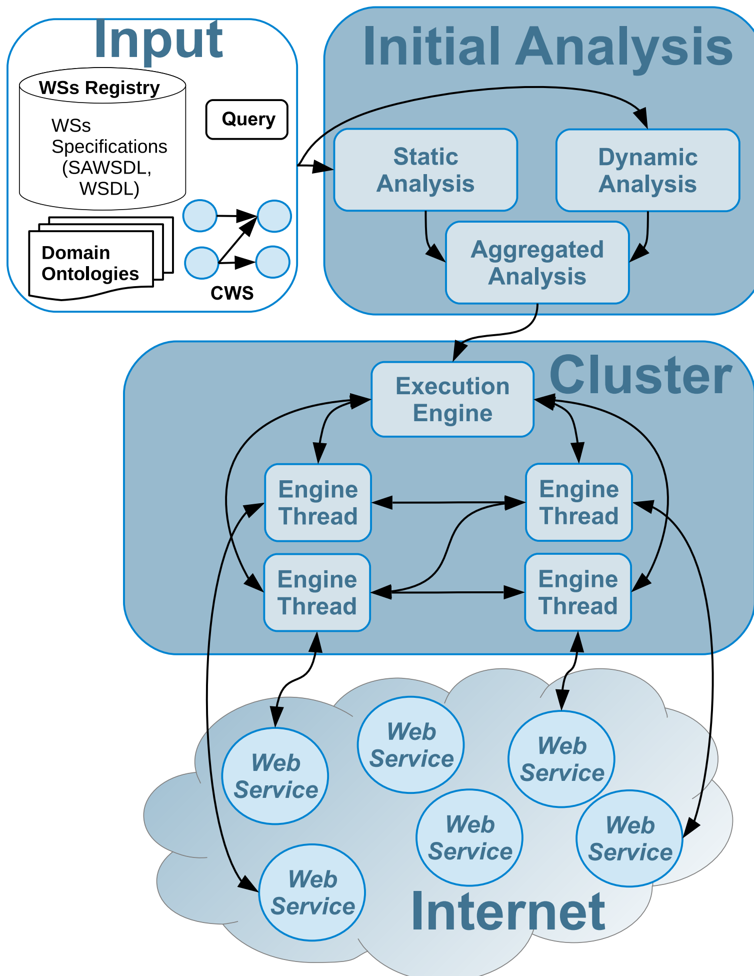

Figure 2 Execution System

Execution state We define the execution state of a CWS in terms of *what happened* and *what remains to happen* at a given moment. For example the elapsed time since the beginning of the execution; how much estimated time remains until the end; how many WSs have been executed; and how many user outputs (O_Q) haven been generated. These parameters are computed during the execution of a CWS as aggregated values while component WSs are executed successfully.

Enviroment state It refers to a set of conditions the whole system has during a CWS execution. These conditions are independent of CWSs and the expected QoS values of its component WSs. We take into consideration the network connectivity parameter as environment state. It is obtained before the CWS execution starts simultaneously for each component WS using a ping measure from the "execution engine" to servers hosting them. Network connectivity is also measured at the moment of selecting a WS replacement.

4 Dynamic recovery decision model

In this Section, we formally describe our model to dynamically choose the best recovery technique. Independently of the technique used for QoS criteria estimation, we assume that each WS is annotated with its estimated execution time, price, reputation, and transactional property.

Definition 4 Estimated Execution Time for a WS (WS_{ETime}). WS_{ETime} represents the estimated execution time for a WS.

Definition 5 Cost for a WS (WS_{COST}). WS_{COST} represents the price that a user has to pay to use the WS and it is fixed by the WS provider.

Definition 6 Reputation for a WS (WS_{REP}). WS_{REP} is an aggregation of users feedbacks and reflects the reliability, trustworthiness and credibility of the service and its provider.

Definition 7 Transactional Property for a WS (TP(WS)). $TP(WS)$ is the transactional property that implicitly describes the behavior of WS in case of failures. It can be pivot (*p*), compensable (*c*), pivot retriable (*pr*), or compensable retriable (*cr*). It depends on the WS developer.

It is possible to calculate the aggregated QoS of a CWS for each criteria in terms of its component WSs. The estimated total execution time is calculated in terms of the component WSs and the execution flow depicted by the structure representing the CWS. In CWSs exist two basic variants of execution scenarios: *sequential execution*, in which the estimated execution time is the sum of estimated execution times of each WS belonging to the sequential path (1), and *parallel execution*, in which the estimated execution time is the maximum estimated execution time of parallel sequential paths (2) [25].

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

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

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} 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. Hence, the total estimated execution time of a CWS is defined as follows:

Definition 8 Estimated Total Execution Time of a CWS (CWS_{ETime}). CWS_{ETime} is the maximum value between all the sequential paths from n_i to n_f. It is calculated using the Bellman-Ford algorithm, whose time complexity is $O(|V||E|)$.

For Cost and Reputation all WSs contribute to the total value independently of the execution flow.

Definition 9 Total Cost of a CWS (CWS_{TCost}). CWS_{TCost} is the sum of all component Web Service costs, defined as: $CWS_{TCost} = \sum_i WS_{i_{COST}}$.

Definition 10 Total Reputation of a CWS (CWS_{TREP}). CWS_{TREP} is the aggregation of all component Web Service reputations, defined as:
$CWS_{TREP} = \prod_i WS_{i_{REP}}$.

Definition 11 Quality associated with a CWS. Let $Q = (I_Q, O_Q, W_Q)$ be the user query and *cws* the CWS which satisfies Q; the quality of *cws* in terms of Q, called $Quality(cws_Q)$, is defined as:

$$Quality(cws_Q) = w_1 * CWS_{ETime} + w_2 * CWS_{TCost} + w_3 * CWS_{TREP} \quad (3)$$

The quality associated to a CWS depends on the QoS criteria and on weights over those criteria. w_1, w_2, and w_3 are the weights for execution time, price, and reputation respectively.

Definition 12 QoS Degree of Fault Tolerance for a CWS ($\Delta QoS(cws)$). $\Delta QoS(cws)$ represents the maximum aggregated value of QoS allowed to exceed for the execution of a CWS. It is expressed as a percentage of $Quality\left(cws_Q\right)$.

$\Delta QoS(cws)$ can be given by the user. In this way, the maximum QoS allowed for a CWS execution is given by its aggregated QoS plus its $\Delta QoS(cws)$:

Definition 13 Tolerated Extra QoS of a CWS ($CWS_{ExtraQoS}$). Let *cws* be a CWS, $Quality(cws_Q)$ its aggregated QoS, and $\Delta QoS(cws)$ its maximum QoS degree supported; $CWS_{ExtraQoS}(cws)$ is defined as:

$$CWS_{ExtraQoS}(cws) = Quality_Q(cws) + \Delta QoS(cws) \quad (4)$$

Definition 14 Real Executed Time for a WS (WS_{RET}). $WS_{i_{RET}}$ refers to the real invested time since ws_i was invoked unti it finishes. If it finishes successfully, it is the time between the moment when it received all its inputs until it sends its produced outputs. In case of failure, it is the time between the moment when it received all its inputs until a failure is detected.

Definition 15 Passed Real Execution Time for a WS (WS_{PT}). Let ws_i be a component WS in a CWS; $WS_{i_{PT}}$ refers to the real invested time since the CWS starts its execution, from n_i, until ws_i is invoked.

With $WS_{i_{PT}}$, 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.

Definition 16 Estimated Remaining Time from a WS ($WS_{RemainT}$). Let ws_i be a component WS in a CWS; $WS_{i_{RemainT}}$ is the maximum value between all the sequential paths from ws_i to n_f. It is calculated as in Definition 8.

$WS_{i_{RemainT}}$ allows to look ahead and calculate *how far* in terms of execution time is the end of a CWS execution respect to each component WS.

Definition 17 Time Degree of Fault Tolerance for a WS ($\Delta Time(ws_i)$). Let *cws* be a CWS with $CWS_{ExtraQoS}(cws)$. Let ws_i be a component WS of *cws* with: $WS_{i_{PT}}$; $WS_{i_{RemainT}}$; $WS_{i_{RET}}$; and $WS_{i_{ETime}}$. $\Delta Time(ws_i)$ represents the maximum time allowed to exceed for the ws_i execution to satisfy $CWS_{ExtraQoS}(cws)$; it is expressed as:

$$\begin{aligned} \Delta Time(ws_i) = \; & CWS_{ExtraQoS_{Time}}(cws) - w_1 * (WS_{iPT}+ \\ & WS_{i_{RemainT}} + WS_{i_{RET}} + WS_{i_{ETime}}) \end{aligned} \quad (5)$$

We can calculate $\Delta Cost(ws_i)$ and $\Delta Rep(ws_i)$ in the same way as we do for $\Delta Time(ws_i)$.

We analyze the network connectivity of each WS to tune up CWS_{ETime}:

Definition 18 Current network connectivity to a WS (WS_{comm}). Let $I(ws_i)$ and $O(ws_i)$ be the inputs and outputs of a WS ws_i; the current network connectivity of ws_i ($WS_{i_{comm}}$) is the estimated transfer time of $I(ws_i)$ and $O(ws_i)$ between the "execution engine" and ws_i.

Hence, we update the estimated execution time for a component WS as:

$$WS_{ETime} = WS_{ETime} + WS_{comm} \quad (6)$$

All the calculations that depend on WS_{ETime} are then also tuned up, such as CWS_{ETime}, $\Delta Time(ws_i)$, and $WS_{RemainT}$.

Finally, we calculate the output dependency of each WS:

Definition 19 Degree of Output Dependency of a WS (WS_{OD}). $WS_{i_{OD}}$ is the number CWS outputs that depend on a successful execution of ws_i. This degree reflects the importance of a WS in terms of the number of user outputs that depends on its successful execution.

4.1 Description of the fault tolerance strategy

Figure 2 shows the steps concerning CWS analysis, which is done before starting the execution. The input is composed by: (i) the CWS to execute, represented as a CWS Graph (Definition 2); (ii) the aggregated QoS of the CWS; (iii) the WS registry containing WS descriptions, their advertised QoS, and their transactional properties; (iv) the query indicating inputs, outputs, and weights over QoS criteria; and (v) the allowed fault tolerance percentage $\Delta QoS(cws)$.

The CWS **Initial Analysis** module is composed by three sub-modules:

(1) **Static Analysis** is responsible for calculating and setting all the CWS properties that can be obtained in a static way, independently of the CWS execution and its environment; these properties are: (i) the aggregated CWS QoS, such as the estimated execution time, price, and reputation; and (ii) the properties regarding each component WS, such as the percentage of user outputs that would not be produced if a WS fails, the remaining estimated execution time, and failure probability from each WS; they are obtained by analyzing the CWS Graph;
(2) **Dynamic Analysis** is in charge of performing the network connectivity verification to each WS in the CWS; it performs the verification by simultaneously sending a ping to each server hosting a WS; and
(3) **Aggregated Analysis** takes the information gathered by the previous analyses and generates a tuned up CWS execution estimation.

Once the CWS initial analysis finishes, we can start the CWS execution. The whole execution is managed by an "execution engine", which deploys one Engine Thread responsible for each WS (as shown in Figure 2). Figure 3 depicts the architecture of an Engine Thread, comprised of the WS Execution, Compliance Monitor, and Engine Thread Fault-tolerance modules.

The Compliance Monitor verifies the QoS constraints before and after a WS execution, determining which one is the best fault-tolerance strategy to perform if a WS fails or if it not possible to continue due to QoS constraint violations.

- **Forward recovery by retrying:** retrying can be immediately executed without considering price and reputation because it is the same ws_i that is re-executed. Execution time is the only QoS criterion considered.
- **Forward recovery with replication or substitution:** it is necessary to consider all QoS criteria of WS substitutes to verify if they satisfy the tolerated extra QoS, $CWS_{ExtraQoS}$. In this case, the aggregated CWS QoS is recalculated considering the replica or substitute of ws_i. It means recalculate (3), let us call $NewQuality(CWS_Q)$, and checking if it does not violate the QoS constraint: $NewQuality(CWS_Q) \leq CWS_{ExtraQoS}$.
- **Backward recovery or checkpointing:** if the QoS constrain can not be satisfied ($NewQuality(CWS_Q) > CWS_{ExtraQoS}$), performing forward recovery is not possible. Thus, we have to determine which one is the best recovery strategy to select between backward recovery and checkpointing. We propose the following weighted sum considering the execution state in terms of elapsed time ($WS_{i_{PT}}$, Definition 15) and the number of outputs obtained ($WS_{i_{OD}}$, Definition 19) despite the failure of ws_i.

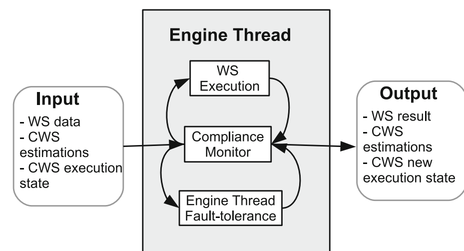

Figure 3 Engine Thread Fault-tolerance

Each recovery strategy has an importance, represented as weights $wback_1$ and $wback_2$, with $\sum_{j=1}^{2} wback_j = 1$, that can be provided by the user query to express preference between backward recovery and checkpointing. We define in (7) a measure to represent the total work done by a CWS at the moment of a failure in terms of elapsed time, produced user outputs, and user preferences.

$$S_i = wback_1 WS_{i_{PT}} + wback_2 WS_{i_{OD}} \tag{7}$$

Thus, while more time is invested since the beginning of a CWS execution and the number of user outputs depending on ws_i is lower, the greater the value of S_i will be. A variable ρ is also defined to specify a threshold of S_i. If $S_i \geq \rho$, the checkpointing strategy is selected to save the work already done; else, backward recovery is executed and all the work is undone.

4.2 Fault tolerant strategy with transactional properties

Let *cws* be a CWS and TP(ws_i) the transactional property of ws_i where: (i) $ws_{pred.ws_i}$ represents a *ws* predecessor of ws_i; (ii) $ws_{succ.ws_i}$ represents a *ws* successor of ws_i; and (iii) $ws_{parall.ws_i}$ a WS executed in parallel with ws_i. The restrictions are:

- *cws* has at most one $\mathrm{TP}(ws_i) = p$. If there is a pivot WS, $\forall ws_j |\ ws_{j_{pred.ws_i}}$, $\mathrm{TP}\big(ws_j\big) = c$ or $\mathrm{TP}\big(ws_j\big) = cr$ to enable backward recovery, i.e., if ws_i fails, all WS predecessors have to be compensated;
- if there exists $\mathrm{TP}(ws_i) = p$ or $\mathrm{TP}(ws_i) = pr$ in *cws*, $\forall ws_j |\ ws_{j_{succ.ws_i}}$, $\mathrm{TP}\big(ws_j\big) = pr$ or $\mathrm{TP}\big(ws_j\big) = cr$ to ensure that everything after ws_i will be executed successfully;
- if there is $\mathrm{TP}(ws_i) = p$, $\forall ws_j |\ ws_{j_{parall.ws_i}}$ $\mathrm{TP}\big(ws_j\big) = cr$, because if ws_i is executed successfully, all its parallel WSs must be also executed successfully because ws_i cannot be compensated, and if ws_i fails, all its parallel WSs must allow compensation;
- if there is $\mathrm{TP}(ws_i) = pr$, $\forall ws_j |\ ws_{j_{parall.ws_i}}$, $\mathrm{TP}\big(ws_j\big) = cr$ or $\mathrm{TP}\big(ws_j\big) = pr$, because if ws_i is executed successfully, all its parallel WSs must be also executed successfully; however, the compensable (*c*) property is not required for its parallel WSs because ws_i will always finish successfully due to its retriable (*r*) property.

The allowed fault tolerance strategies according to the transactional property of a faulty WS are summarized in Table 1. For instance, line 1 shows that regardless the transactional property of the faulty WS, if it has a predecessor or parallel WS with transactional property *p*, only forward recovery or checkpointing can be selected. If replication or substitution are applied, replicas or substitutes of the faulty ws_i have to satisfy QoS constraints and transactional restrictions. It means that transactional properties of replicas or substitute WS have to be the same of TP(ws_i) or one that does not violate the CWS transactional property.

5 Algorithms for fault tolerant CWS execution

Algorithm 1 shows CWS Initial Analysis. Lines 1 and 2 calculate CWS static values; that is, values that do not change with environment conditions. Network connectivity is evaluated for each WS in the CWS, and their WS_{ETime} is updated (lines 4 to 5). Note that the instruction of line 4 represents the process to check WSs connectivity. If the WS is not responding, then a WS substitution should be done. If there is no substitute, a CWS reconfiguration

Table 1 Recovery techniques when transactional properties are considered in a CWS

Failed TP(ws_i)	TP($ws_{pred.ws_i}$)	TP($ws_{parall.ws_i}$)	Decision
Any	*p*	*p*	*fr* or *ckp*
p, *pr*	*c*, *cr*	*cr*	any
pr	*pr*	*pr*	*fr* or *ckp*
c	*pr*	any	*fr* or *ckp*
c, *cr*	*c*, *cr*	*c*, *cr*	any
cr	*p*, *pr*	*p*, *pr*	*fr* or *ckp*

should be tried, or else abort the CWS execution. Finally, nodes are annotated with their information (line 6), and CWS_{ETime} is updated (line 7).

Algorithm 1 CWS Initial Analysis

```
   Input: cws, Q, ΔQoS(cws)
   Output: cwsestimations, Quality(cws_Q)
   begin
1      set CWS_ETime, CWS_TCost, CWS_TREP;
2      set Quality(cws_Q);
3      set CWS_Extra_QoS(cws); /*The following loop should be performed in parallel*/
       for ws ∈ CWS do
4          ws_comm ← ping ws;
           /*if the WS does not respond, further actions should be taken (e.g.,
           substitution) */
5          ws_ETime ← ws_ETime + ws_Comm;
6          set ws_RemainT,ws_OD to ws; /*Set individual node information*/
7      set CWS_ETime /*Tuned up*/
```

Algorithm 2 shows the execution control for a WS invocation. As a preventive strategy, we propose a CWS analysis to identify critical WSs which can exceed the time constraints if they fail; hence, WSs can be replicated at the moment of their first invocation to improve their probability of success. A WS can also be replicated if during the CWS execution, it has become a critical WS (e.g., due to previous failures). When a WS is going to be executed, the Engine Thread checks the strategy to be performed by calling Algorithm 4 (line 1). If the strategy is *replicate*, *retry*, or *none*, it calls Algorithm 3 to do the WS execution (line 2). Algorithm 3 is responsible for WS executions. It can perform replication (line 1) or single WS execution (line 2).

Algorithm 2 Engine Thread

```
   Input: WS_i (the WS and all its related
   data)
   begin
       repeat
1          strategy ← call Algorithm 4;
           if strategy ==
           (replicate ∨ retry ∨ ∅) then
2              success ← call Algorithm 3;
3      until (success ∨ strategy =
       (backward_rec ∨ checkpointing));
```

Algorithm 3 WS Execution

```
    Input: strategy, WS_i
    Output: success
    begin
        if strategy == replicate then
1           success ← replicate WS_i;
        else
2           success ← invoke WS_i;
        return success;
```

Algorithm 4 is responsible for selecting the best recovery strategy. It receives as input the global parameters of the CWS execution (line 1), and the parameter concerning the analyzed WS (2). It first checks for the possibility of performing forward recovery (line 3). If there is enough time to perform forward recovery, it checks if the WS must be replicated as prevention, retried, or substituted (line 4). If there is no need of replication, then it checks if the WS can be retried (line 5) or substituted (line 6). If forward recovery is not possible, it checks the amount of work done by the CWS to decide between backward recovery and checkpointing (line 8).

Algorithm 4 Fault tolerance strategy selection

```
1   Input: CWS_ETime, wback_1, wback_2, ρ, CWS_ExtraQoS(cws) /*global parameters*/
2   Input: WS_iPT, WS_iRemainT, WS_iRET, WS_iETime, WS_iOD /*WS parameters*/
    Output: fault tolerance strategy
    begin
3       if ΔQoS_(ws_i) > 0 then
            /*forward recovery: retry, replication, substitution*/
4           if ΔTime_(ws_i) − WS_iETime < 0 ∧
            ∃ws_r|ws_r satisfies CWS_ExtraQoS(cws) ∧ ws_r replaces WS_i then
                strategy ← replicate; /*preventive*/
            else
5               if TP(WS) == (pr ∨ cr) then
                    strategy ← retry;
                else
6                   if ∃ws_r|ws_r satisfies CWS_ExtraQoS(cws) ∧ ws_r replaces WS_i then
                        strategy ← substitution;

7       if strategy == ∅ then
            set S_i with wback_1 and wback_2 (see Eq. 7)
8           if S_i ≥ ρ then
                strategy ← checkpointing;
            else
                strategy ← backward_recovery;
        return strategy;
```

6 Related work

Several techniques have been proposed to implement reliable CWS execution. Some works consider WS transactional properties to ensure the all-or-nothing property of CWSs [7, 9, 11, 13, 24]. In this context, failures during CWS executions can be repaired by backward or forward recovery processes. Other works consider WS replication, instead of transactional properties, to provide forward recovery [4, 18, 29]. For some queries, partial responses may

have sense; then, checkpointing techniques can be implemented to relax the all-or-nothing property and still provide fault tolerance [6, 20, 22]. The faulty CWS can be resumed from an advanced execution point to complete the desired result. None of these works consider the dynamism of the execution environment to adapt the decision regarding to which recovery strategy is the most appropriate.

Regarding selfhealing approaches, some works build on top of BPEL [3, 15, 17, 23], while others propose new engines [12, 16, 27]. Modafferi and Conforti [15] present an approach where developers define a Ws-BPEL process annotated with recovery information. This Ws-BPEL process is then transformed in a standard Ws-BPEL process. The supported recovery mechanisms are: external variable setting; timeout; redo; future alternate behaviour; and rollback and re-execution. Moser et al. [17] present a system to monitor BPEL processes regarding QoS constraints. It allows: the adaptation of existing processes by providing alternative services for a given service; and the transformation of SOAP messages to handle service interface mismatches. It is implemented using AOP, decoupling it from the BPEL engine. Baresi et al. [3] augment the BPEL technology with supervision rules to set what to check at runtime, and to define how to act when anomalies are found. Subramanian [23] et al. propose an extension to BPEL regarding self-healing policies. It allows definition of pre- and post-conditions of BPEL activities; monitoring; diagnosis; and recovery strategy suggestion. Halima et al. [12] propose a self-healing framework based on QoS. It enhances the messages between WSs with QoS metadata. This QoS metada is used to detect QoS degradation, and react accordingly (e.g., WS substitution). Moo-Mena et al. [16] propose a QoS approach to duplicate or substitute WSs in case of QoS degradation. Reponses between WSs are intercepted to check if there is a SLA degradation. Zheng et al. [27] define an adaptive and dynamic fault tolerance strategy based on execution time, failures probability, and resource consumption parameters. Users specify weights over those parameters to help choosing the recovery strategy that complies with its needs. This last approach is meant for single WS executions, not CWSs.

Our work proposes a non intrusive self-healing CWS execution framework, which is transparent to users and developers. We introduce a novel approach to measure the work done by a CWS execution and its compliance with QoS requirements. The CWS work is expressed in terms of expected, current, and remaining QoS, and expected and produced user outputs. This measure supports the selection of the most appropiate recovery or preventive strategy. As far as we know, existing work lack this kind of work measure for CWSs, as well as the dynamism regarding fault tolerance and preventive strategies. The main disadvantage of our approach is that it is a new engine that does not build on top of accepted standards, such as BPEL; however, it is conceived as an automatic CWS execution engine expected to work with the least amount of human intervention possible. Nonetheless, the main concepts of our solution can be implemented as a complement of any other self-healing solution for CWSs.

7 Experimental study

7.1 Implementation and general setup

We developed an execution engine using Java 6 and the MPJ Express 0.38 library. We deployed it 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

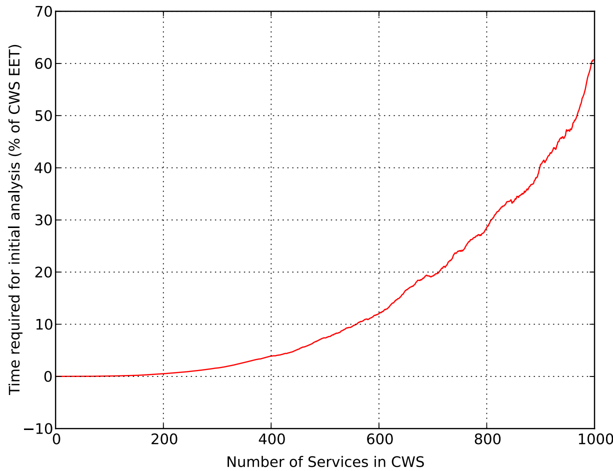

Figure 4 CWS Initial Analysis

CPU, 1GB RAM, Debian 6.0. They are connected through a 100Mbps Ethernet. We generate 1000 CWSs consisting of 1 to 1000 WSs using the Barabási-Albert model [2]. All WSs, including replicas, have different QoS values. We consider the following QoS parameters: estimated execution time; availability; cost; and reputation. We took real QoS values from WS-DREAM [28]. All used artifacts are available[2].

7.2 Efficiency evaluation

We evaluate the efficiency of our approach in terms of the performance of the CWS Initial Analysis, since it is the most time consuming operation in our system. The CWS Initial Analysis module depends on the CWS Graph analysis to obtain values such as CWS_{ETime}, CWS_{TCost}, CWS_{TREP}, $WS_{RemainT}$, and WS_{OD}; hence, its running time depends on the size and complexity of the CWS Graph. Individual WS monitoring does not depend on the size of the graph, so it does not have relevant impact on the performance of our system.

We measure the efficiency of the CWS Initial Analysis as as a percentage of the CWS_{ETime} of the CWS in evaluation. Figure 4 shows that the time consumed to analyse CWSs with less than 400 WSs is relatively low: below 5 % of CWS_{ETime}. The analysis time for larger CWSs is higher in relation to CWS_{ETime}: around 60 % of the CWS_{ETime} for CWSs containing 1000 WSs.

Regarding network connectivity in the Dynamic Analysis, if we suppose that we have the capacity of performing the parallel verification of all WSs in a CWS, it takes an average of 88.18 ms to get a reponse from WSs servers. The maximum value for a reponse we got from evaluating all the 1,000,000 servers of the dataset was 483.23 ms, while 0.29 % were not available.

[2] http://www.lamsade.dauphine.fr/~angarita/des.html

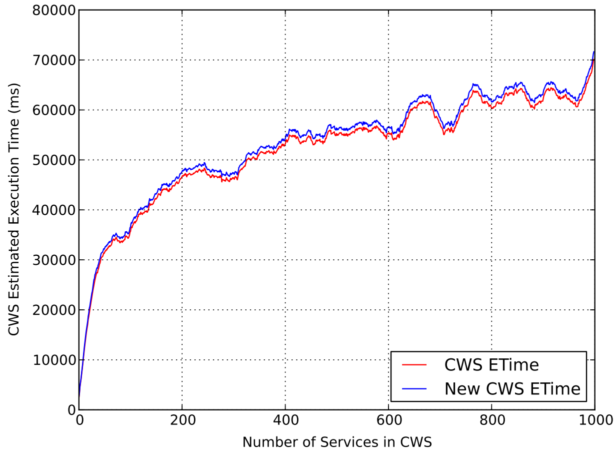

Figure 5 Tuned up CWS_{ETime}

7.3 Effectiveness evaluation

We start the effectiveness evaluation by performing the CWS Initial Analysis, wich produces a tuned up CWS_{ETime}. Figure 5 shows the original CWS_{ETime} (CWS ETime) and the new tuned up CWS_{ETime} (New CWS ETime), taking into accout the network connectivity at the moment of CWS executions. Some servers were unavailable; therefore, a CWS reconfiguration (e.g., WS replacement) would have to be done before executing the CWS.

We have choosen a CWS (Figure 6) among the generated ones to illustrate our approach. Arcs between WSs represent the data flow or control flow relations; arc numbers between WSs and n_f indicate the number of user outputs produced by its corresponding WS. Table 2

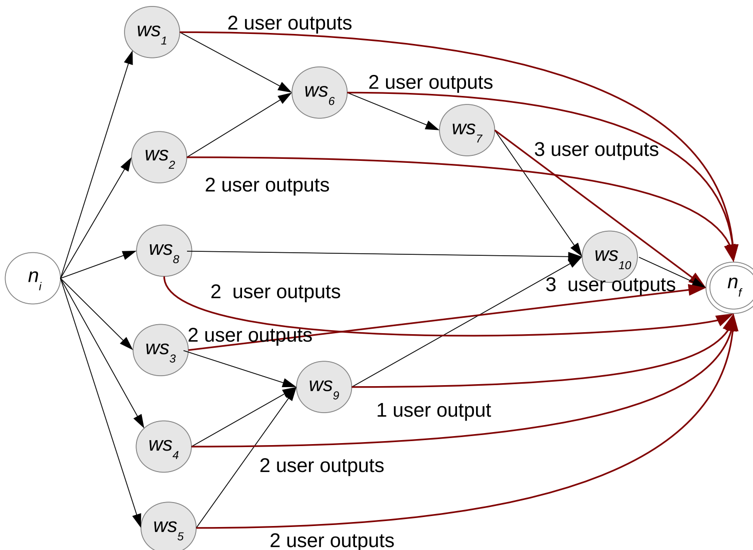

Figure 6 Ilustrative CWS

Table 2 WSs QoS and output degree

component WS	WS_{ETime} (secs)	WS_{COST}	WS_{REP}	WS_{OD} (#)
ws_1	8080	80	0.9	10
ws_2	8020	85	0.8	10
ws_3	34980	0	0.9	6
ws_4	7570	0	0.7	6
ws_5	12990	75	0.8	6
ws_6	836	90	0.9	8
ws_7	1388	73	0.9	6
ws_8	13330	0	0.7	5
ws_9	24720	0	0.8	4
ws_{10}	29650	81	0.9	3

shows the WS_{ETime} (in seconds), WS_{COST}, WS_{REP}, and the WS_{OD} for each component WS of our example CWS. Thus, we have the following values for the CWS:

$$CWS_{ETtime} = WS_{ETtime_{ws_3}} + WS_{ETtime_{ws_9}} + WS_{ETtime_{ws_{10}}} = 89350 secs$$

$$CWS_{TCost} = \sum_{i=1}^{10} WS_{i_{COST}} = 484;\ CWS_{TREP} = \prod_{i=1}^{10} ws_{i_{REP}} = 0.14$$

$$Supposethat\ w_1 \equiv w_2 \equiv w_3, \Delta QoS(cws) = 30\%,\ and\ \rho = 50\%.$$

Case 1 forward recovery by retrying: a retriable WS already satisfies cost and reputation. We have to verify if there is time for retrying. Suppose that $TP(ws_9) = r$, and that ws_9 fails after 24700 secs; thus, we have that:

- $WS_{PT_{ws_9}}$ = 34980 secs;
- $WS_{RemainT_{ws_9}}$ = 54370 secs $\left(WS_{ETime_{ws_9}} + WS_{ETime_{ws_{10}}}\right)$;
- $WS_{ETime_{ws_9}}$ = 24700 secs.

We have that the new CWS_{ETime} = 114050 secs, the original CWS_{ETime} was 89350 secs; representing an increment of the 27.64 %. The user allows a 30 % extra for the execution time, $\Delta Time(ws_9) > 0$, so ws_9 can be reexecuted.

Case 2 forward recovery with replication or substitution: now, suppose that we are in the same situation of case 1; however, $TP(ws_9) \neq r$. What can we do? We look at the possible WS substitutes that satisfy:

$$NewQuality(CWS_Q) \leq CWS_{ExtraQos}$$

If there is not much time for a WS substitute execution, we can replicate a set of substitute WSs such that their parallel execution satisfy the above equation, taking the results of the first one ending successfully (Figure 7). Note that for substitute WSs we have to consider all QoS criteria.

Case 3 backward recovery or checkpointing: suppose that $TP(ws_9) \neq r$, and it has no substitutes; so $\Delta Time(ws_i) < 0$ or $NewQuality(CWS_Q) > CWS_{ExtraQoS}$. In this case,

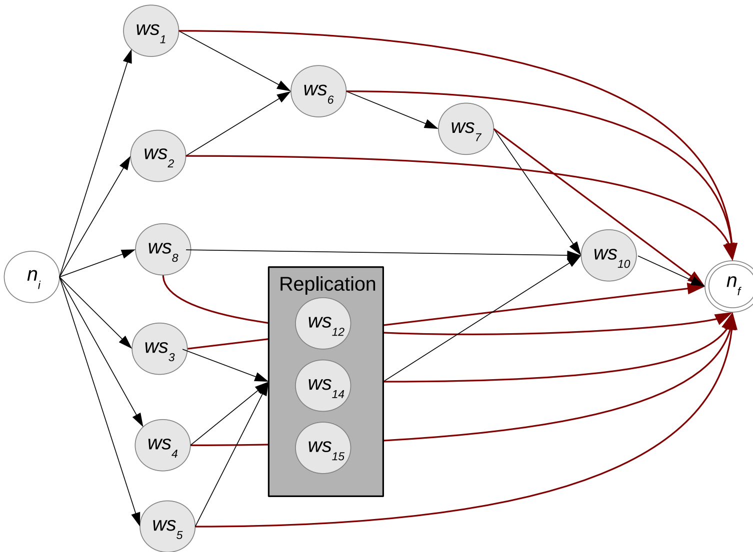

Figure 7 Substitution of ws_9 with replication

new CWS_{ETime} = 114050 (see case 1) and $\Delta Time(ws_9) < 0$; therefore, forward recovery cannot be selected. We know the number of user outputs depending on the successful execution of ws_9 (Table 6), therefore the value of WS_{OD} can be calculated, representing 80.95% of the total user outputs (see Definition 19).

$$S_{ws_9} = w_1 WS_{PT_{ws_9}} + w_2 WS_{OD_{ws_9}} = 0.5(66.79) + 0.5(80.95) = 73.87$$

Then, since $S_{ws_9} \geq \rho$, the selected strategy is checkpointing. We suppose the rest of WSs until ws_7 were executed successfully. ws_{10} received a correct output from ws_7 and a checkpointing message from s_{ws_9}, so it cannot be executed but skipped. At the end of the execution, the CWS would have generated all the user outputs except the ones generated by ws_9 and ws_{10}. The partial outputs delivered to the user represent 80.95 % of the total outputs. The total executed time would be $WS_{PT_{ws_9}}$ + 24700 seconds = 59680 seconds and the remaining time in case of execution restart would be the execution times of ws_9 and ws_{10} (Figure 8), which is 54370 seconds. Those amounts of time represent 66.79 % and 60.85 % of the total estimated execution time, respectively. The new QoS criteria are: $newCWS_{ETime} = 114050$, $newCWS_{TCost} = CWS_{TCost} + WS_{COST_{w_9}}$, $newCWS_{REP} = CWS_{REP} \cdot WS_{REP_{WS_9}}$.

7.4 CWS executions effectiveness evaluation

We propose two environments with low WS availability: availability = 0.6, and availability = 0.8 per WS of Figure 6, producing low availability CWS. We also set 0 as a tolerance for CWS extra execution time, but we let the CWS cost and reputation open to any extra value. Figure 7 shows results of performing 100 executions of the CWS of Figure 6 with 0,1,2,3,4,5, and 6 replicas. Regarding time constraint violation, it can be observed how it decreases using availability = 0.6 (Time Violation 0.6), and availability = 0.8 (Time Violation 0.8). For the worst availability (0.6), we have that, without using replication, the time constraint is violated in 100 % of the cases, due to the need of performing constant WSs retries, or substitutions. The same behaviour can be observed using WSs availability of 0.8,

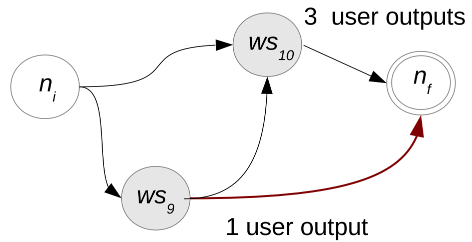

Figure 8 Checkpointing/resume

but with a lower rate of time constraint violation. Figure 9 shows how the global cost of the CWS increases when using replication for all WSs to improve the success rate as much as possible (Cost All WSs), and using replication only for WSs in the critical path (Cost critical path) : ws_3, ws_9, and ws_{10}. We represent cost as the additional WSs that were successfully executed due to of replication.

8 Conclusions

We have extended our approach proposed in [1] to enable the selection of the most appropriate recovery strategy in a dynamic way, according to QoS constraints, context and environment information, such as the execution state and progress of a CWS and the network connectivity at the exact moment of the CWS execution of each WS in the CWS. The considered recovery strategies backward and forward recovery based on transactional properties. Forward recovery can be performed by retrying, replication, and substitution. We use

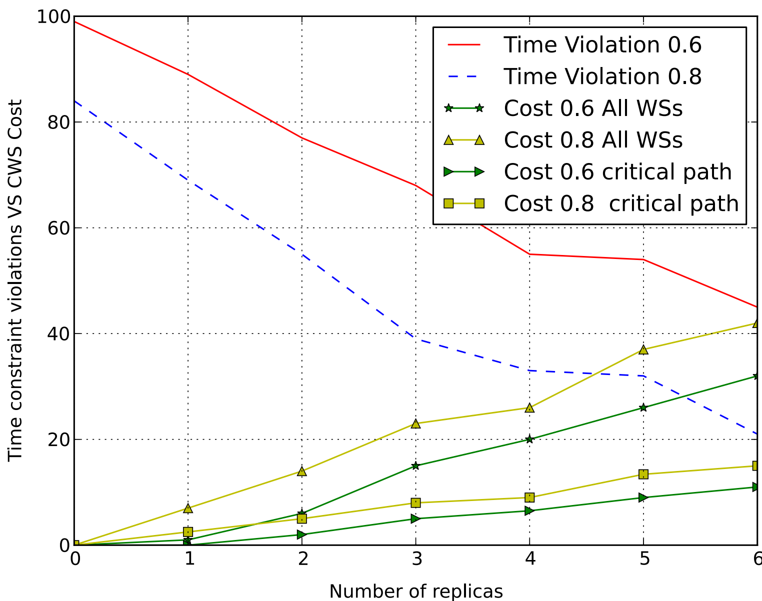

Figure 9 Constraint violations and cost

replication also as a preventive strategy, and checkpointing as an alternative strategy. We have showed the performance of our approach using large CWSs, and the impact on the CWS QoS of using replication to decrease the time constraint violation rate. Regarding our future work, the next task will be to make available a user friendly implementation of our system.

References

1. Angarita, R., Cardinale, Y., Rukoz, M.: Dynamic recovery decision during composite web services execution. In: Proceedings of the Fifth Int. Conf. on Mngmt. of Emergent Digital EcoSystems, pp. 187–194. ACM (2013)
2. Barabási, A.-L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
3. Baresi, L., Guinea, S.: Dynamo and self-healing bpel compositions. In: 29th International Conference on Software Engineering - Companion, 2007 ICSE, 2007 Companion, pp. 69–70 (2007)
4. Behl, J., Distler, T., Heisig, F., et al.: Providing Fault-tolerant Execution of Web-service based Workflows within Clouds. In: Proceedings of the 2nd Int. Workshop on Cloud Computing Platforms (CloudCP) (2012)
5. Benjamins, R., Dorner, J.D.E., Domingue, J., Fensel, D., López, O., Volz, R., Wahler, A., Zaremba, M.: Service web 3.0. Technical report, Semantic Technology Institutes International (2007)
6. Brzezinski, J., Danilecki, A., Holenko, M., Kobusinska, A., Kobusinski, J., Zierhoffer, P.: D-reserve: Distributed reliable service environment. In: ADBIS, pp. 71–84 (2012)
7. Bushehrian, O., Zare, S., Rad, N.K.: A Workflow-Based Failure Recovery in Web Services Composition. J. Softw. Eng. Appl. **5**, 89–95 (2012)
8. Business Process Execution Language for Web Services (bpel4ws), 2001. http://www.ibm.com/developerworks/library/specification/ws-bpel/ - Extracted on April 2012
9. Cardinale, Y., Rukoz, M.: A framework for reliable execution of transactional composite web services. In: Proceedings of The Int. ACM Conf. on Mngmt. of Emergent Digital EcoSystems (MEDES), pp. 129–136 (2011)
10. Chan, K., Bishop, J., Steyn, J., Baresi, L., Guinea, S.: A fault taxonomy for web service composition. In: Service-Oriented Computing - ICSOC 2007 Workshops, vol. 4907 of *Lecture Notes in Computer Science*, pp. 363–375. Springer, Berlin Heidelberg (2009)
11. Haddad, J.E., Manouvrier, M., Rukoz, M.: TQoS: Transactional and QoS-aware selection algorithm for automatic Web service composition. IEEE Trans. Serv. Comput. **3**(1), 73–85 (2010)
12. Halima, R.B., Drira, K., Jmaiel, M.: A qos-oriented reconfigurable middleware for self-healing web services. In: Proceedings of the 2008 IEEE International Conference on Web Services, ICWS '08, pp. 104–111, Washington DC, USA, 2008. IEEE Computer Society
13. Lakhal, N.B., Kobayashi, T., Yokota, H.: FENECIA: failure endurable nested-transaction based execution of composite Web services with incorporated state analysis. VLDB J. **18**(1), 1–56 (2009)
14. Liu, A., Li, Q., Huang, L., Xiao, M.: FACTS: A framework for fault tolerant composition of transactional web services. IEEE Trans. Serv. Comput. **3**(1), 46–59 (2010)
15. Modafferi, S., Conforti, E.: Methods for enabling recovery actions in ws-bpel. In: Proceedings of the 2006 Confederated Int. Conf. on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I, pp. 219–236. Springer, Berlin, Heidelberg (2006)
16. Moo-Mena, F., Garcilazo-Ortiz, J., Basto-Díaz, L., et al.: Defining a self-healing qos-based infrastructure for web services applications. In: Proceedings of the 2008 11th IEEE Int. Conf. on Comp. Sci. and Eng. - Workshops, pp. 215–220. IEEE Computer Society, Washington (2008)
17. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for ws-bpel. In: Proceedings of the 17th International Conference on World Wide Web, WWW '08, pp. 815–824. ACM, New York (2008)
18. Nascimento, A.S., Rubira, C.M.F., Burrows, R., et al.: A systematic review of design diversity-based solutions for fault-tolerant soas. In: Proceedings of Int. Conf. on Eval. and Assessment in Software Eng., pp. 107–118 (2013)
19. OASIS: Web Services Businnes Process Execution Language (WS-BPEL), Version 2.0. OASIS Standard (2007). http://docs.oasis-open.org/wsbepel/2.0/wsbpel-v2.0.html,2007.OASISStandard
20. Rukoz, M., Cardinale, Y., Angarita, R.: Faceta*: Checkpointing for transactional composite web service execution based on petri-nets. Procedia Comput. Sci. **10**, 874–879 (2012)

21. Schafer, M., Dolog, P., Nejdl, W.: An environment for flexible advanced compensations of web service transactions. ACM Trans. Web, 2 (2008)
22. Sindrilaru, E., Costan, A., Cristea, V.: Fault tolerance and recovery in grid workflow management systems. In: Interl Conf. on Complex, Intelligent and Software Intensive Systems, pp. 475–480 (2010)
23. Subramanian, S., Thiran, P., Narendra, N.C., et al.: On the enhancement of bpel engines for self-healing composite web services. In: Proceedings of the 2008 Int. Symposium on Applications and the Internet, SAINT '08, pp. 33–39. IEEE Computer Society, Washington (2008)
24. Wu, Q., Zhu, Q.: Transactional and qos-aware dynamic service composition based on ant colony optimization. Future Gener. Comput. Syst. **29**(5), 1112–1119 (2013)
25. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: Qos-aware middleware for web services composition. IEEE Trans. Softw. Eng. **30**(5), 311–327 (2004)
26. Zhao, Z., Wei, J., Lin, L., et al.: A Concurrency Control Mechanism for Composite Service Supporting User-Defined Relaxed Atomicity. In: The 32nd IEEE Int. Computer Soft. and App. Conf., pp. 275–278 (2008)
27. Zheng, Z., Lyu, M.R.: An adaptive qos-aware fault tolerance strategy for web services. Empirical Softw. Engg. **15**(4), 323–345 (2010)
28. Zheng, Z., Lyu, M.: Collaborative reliability prediction of service-oriented systems. In: Conf. on Software Engineering, 2010 ACM/IEEE 32nd Int., vol. 1, pp. 35–44 (2010)
29. Zhou, W., Wang, L.: A byzantine fault tolerant protocol for composite web services. In: International Conference on Computational Intelligence and Software Engineering (CiSE), pp. 1–4 (2010)