FaCETa: Backward and Forward Recovery for Execution of Transactional Composite WS*

Rafael Angarita[1], Yudith Cardinale[1], and Marta Rukoz[2]

[1] Departamento de Computación, Universidad Simón Bolívar,
Caracas, Venezuela 1080
[2] LAMSADE, Université Paris Dauphine
Université Paris Ouest Nanterre La Défense
Paris, France
{yudith,rangarita}@ldc.usb.ve, marta.rukoz@lamsade.dauphine.fr

Abstract. In distributed software contexts, Web Services (WSs) that provide transactional properties are useful to guarantee reliable Transactional Composite WSs (TCWSs) execution and to ensure the whole system consistent state even in presence of failures. Fails during the execution of a TCWS can be repaired by forward or backward recovery processes, according to the component WSs transactional properties. In this paper, we present the architecture and an implementation of a framework, called FaCETa, for efficient, fault tolerant, and correct distributed execution of TCWSs. FaCETa relies on WSs replacement, on a compensation protocol, and on unrolling processes of Colored Petri-Nets to support fails. We implemented FaCETa in a Message Passing Interface (MPI) cluster of PCs in order to analyze and compare the behavior of the recovery techniques and the intrusiveness of the framework.

1 Introduction

Large computing infrastructures, like Internet increase the capacity to share information and services across organizations. For this purpose, Web Services (WSs) have gained popularity in both research and commercial sectors. Semantic WS technology [20] aims to provide for rich semantic specifications of WSs through several specification languages such as OWL for Services (OWL-S), the Web Services Modeling Ontology (WSMO), WSDL-S, and Semantic Annotations for WSDL and XML Schema (SAWSDL) [15]. That technology supports WS composition and execution allowing a user request be satisfied by a Composite WS, in which several WSs and/or Composite WSs work together to respond the user query.

WS Composition and the related execution issues have been extensively treated in the literature by guaranteeing user *QoS* requirements and fault tolerant execution [7, 11, 16, 18, 21]. WSs that provide transactional properties are useful to guarantee reliable Transactional Composite WSs (TCWSs) execution, in order to ensure that the whole system remains in a consistent state

* This work was supported by the Franco-Venezuelan CNRS-FONACIT project N°22782

even in presence of failures. TCWS becomes a key mechanism to cope with challenges of open-world software. Indeed, TCWS have to adapt to the open, dynamically changing environment, and unpredictable conditions of distributed applications, where remote services may be affected by failures and availability of resources [27].

Generally, the control flow and the order of WSs execution is represented with a structure, such as workflows, graphs, or Petri-Nets [3, 6, 7, 14]. The actual execution of such TCWS, carried out by an EXECUTER, could be deployed with centralized or distributed control. The EXECUTER is in charge of *(i)* invoking actually WSs for their execution, *(ii)* controlling the execution flow, according to data flow structure representing the TCWS, and *(iii)* applying recovery actions in case of failures in order to ensure the whole system consistence; fails during the execution of a TCWS can be repaired by forward or backward recovery processes, according to the component WSs transactional properties.

In previous works [9, 10] we formalized a fault tolerant execution control mechanism based on Colored-Petri Nets (`CPN`), which represent the TCWS and the compensation process. In [9] unrolling algorithms of `CPNs` to control the execution and backward recovery were presented. This work was extended in [10] to consider forward recovery based on WS replacement; formal definitions for WSs substitution process, in case of failures, were presented. In [10], we also proposed an EXECUTER architecture, independent of its implementation, to execute a TCWS following our proposed fault tolerant execution approach.

In this paper, we present an implementation of our EXECUTER framework, called FACETA (FAult tolerant Cws Execution based on Transactional properties), for efficient, fault tolerant, and correct distributed execution of TCWSs. We implemented FACETA in a Message Passing Interface (MPI) cluster of PCs in order to analyze and compare the efficiency and performance of the recovery techniques and the intrusiveness of the framework. The results show that FACETA efficiently implements fault tolerant strategies for the execution of TCWSs with small overhead.

2 TCWS Fault-Tolerant Execution

This Section recall some important issues related to transactional properties and backward and forward recovery, presented in our previous works [7, 9, 10]. We consider that the Registry, in which all WSs are registered with their corresponding *WSDL and OWLS documents*, is modeled as a Colored Petri-Net (`CPN`), where WS inputs and outputs are represented by places and WSs, with their transactional properties, are represented by colored transitions - colors distinguish WS transactional properties [7]. The `CPN` representing the Registry describes the data flow relation among all WSs.

We define a query in terms of functional conditions, expressed as input and output attributes; *QoS* constraints, expressed as weights over criteria; and the required global transactional property as follows. A Query Q is a 4-tuple (I_Q, O_Q, W_Q, T_Q), where:

- I_Q is a set of input attributes whose values are provided by the user,

- O_Q is a set of output attributes whose values have to be produced by the system,
- $W_Q = \{(w_i, q_i) \mid w_i \in [0,1]$ with $\sum_i w_i = 1$ and q_i is a QoS criterion$\}$, and
- T_Q is the required transactional property; in any case, if the execution is not successful, nothing is changed on the system and its state is consistent.

A TCWS, which answers and satisfies a Query Q, is modeled as an acyclic marked CPN, called CPN-$TCWS_Q$, and it is a sub-net of the Registry CPN[1]. The *Initial Marking* of CPN-$TCWS_Q$ is dictated by the user inputs. In this way, the execution control is guided by a unrolling algorithm.

2.1 Transactional Properties

The transactional property (TP) of a WS allows to recover the system in case of failures during the execution. The most used definition of individual WS transactional properties ($TP(ws)$) is as follows [8, 13]. Let s be a WS: s is **pivot** (p), if once s successfully completes, its effects remains forever and cannot be semantically undone (compensated), if it fails, it has no effect at all; s is **compensatable** (c), if it exists another WS s', which can semantically undo the execution of s, even after s successfully completes; s is **retriable** (r), if s guarantees a successfully termination after a finite number of invocations; the **retriable** property can be combined with properties p and c defining **pivot retriable** (pr) and **compensatable retriable** (cr) WSs.

In [11] the following TP of TCWS have been derived from the TP of its component WSs and their execution order(sequential or parallel). Let tcs be a TCWS: tcs is **atomic** ($\boldsymbol{a}$), if once all its component WSs complete successfully, they cannot be semantically undone, if one component WS does not complete successfully, all previously successful component WSs have to be compensated; tcs is **compensatable** (c), if all its component WSs are compensatable; tcs is **retriable** (r), if all its component WSs are retriable; the retriable property can be combined with properties $\boldsymbol{a}$ and c defining **atomic retriable** ($\boldsymbol{a}r$) and **compensatable retriable** (cr) TCWSs.

According to these transactional properties, we can establish two possible recovery techniques in case of failures:

- *Backward* recovery: it consists in restoring the state (or a semantically closed state) that the system had at the beginning of the TCWS execution; i.e., all the successfully executed WSs, before the fail, must be compensated to undo their produced effects. All transactional properties (p, $\boldsymbol{a}$, c, pr, $\boldsymbol{a}r$, and cr) allow backward recovery;
- *Forward* recovery: it consists in repairing the failure to allow the failed WS to continue its execution. Transactional properties pr, $\boldsymbol{a}r$, and cr allow forward recovery.

[1] A marked CPN is a CPN having tokens in its places, where tokens represent that the values of attributes (inputs or outputs) have been provided by the user or produced by a WS execution.

2.2 Backward Recovery Process: unrolling a Colored Petri-Net

The global TP of CPN-$TCWS_Q$ ensures that if a component WS, whose TP does not allow forward recovery fails, then all previous executed WSs can be compensated by a backward recovery process. For modeling TCWS backward recovery, we have defined a backward recovery `CPN`, called BRCPN-$TCWS_Q$, associated to a CPN-$TCWS_Q$ [9]. The component WSs of BRCPN-$TCWS_Q$ are the compensation WSs, s', corresponding to all c and cr WSs in CPN-$TCWS_Q$. The BRCPN-$TCWS_Q$ represents the compensation flow, which is the inverse of the execution order flow. In BRCPN-$TCWS_Q$ a color of a transition s' represents the execution state of its associated transition s in the CPN-$TCWS_Q$ and is updated during CPN-$TCWS_Q$ execution. Color(s') $\in$ {I='initial', R='running', E='executed', C='compesated', A='Abandonned'}thus, if color(s')='E' means that its corresponding WS s is being executed. In [7, 9] we propose techniques to automatically generate both `CPNs`, CPN-$TCWS_Q$ and BRCPN-$TCWS_Q$.

The execution control of a TCWS is guided by a unrolling algorithm of its corresponding CPN-$TCWS_Q$. A WS is executed if all its inputs have been provided or produced, i.e., each input place has as many tokens as WSs produce them or one token if the user provide them. Once a WS is executed, its input places are unmarked and its output places (if any) are marked.

The compensation control of a TCWS is also guided by a unrolling algorithm. When a WS represented by a transition s fails, the unrolling process over CPN-$TCWS_Q$ is halted,an *Initial Marking* on the corresponding BRCPN-$TCWS_Q$ is set (tokens are added to places associated to input places of the faulty WS s, and to places representing inputs of BRCPN-$TCWS_Q$, i.e., places without predecessors) and a backward recovery is initiated with the unrolling process over BRCPN-$TCWS_Q$. We illustrate a backward recovery in Figure 1. The marked CPN-$TCWS_Q$ depicted in Figure 1(a) is the state when ws_4 fails, the unrolling of CPN-$TCWS_Q$ is halted, and the *Initial Marking* on the corresponding BRCPN-$TCWS_Q$ is set to start its unrolling process (see Figure 1(b)), after ws'_3 and ws'_5 are executed and ws_7 is abandoned before its invocation, a new *Marking* is produced (see Figure 1(c)), in which ws'_1 and ws'_2 are both ready to be executed and can be invoked in parallel. Note that only **compensatable** transitions have their corresponding compensation transitions in BRCPN-$TCWS_Q$.

2.3 Forward Recovery Process: Execution with WS Substitution

During the execution of TCWSs, if a failure occurs in an advanced execution point, a backward recovery may incur in high wasted resources. On the other hand, it is hard to provide a **retriable** TCWS, in which all its components are **retriable** to guaranty forward recovery. We proposed an approach based on WS substitution in order to try forward recovery [10]. TCWS composition and execution processes deal with *service classes* [1], which group WSs with the same semantic functionality, i.e., WSs providing the same operations but having different WSDL interfaces (input and output attributes), transactional support, and QoS. When a WS fails, if it is not **retriable**, instead of backward recovery, an substitute WS is searched to be executed on behalf of the faulty WS.

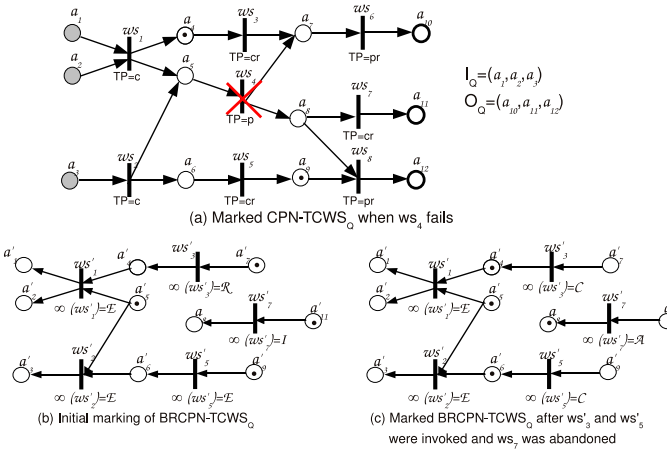


Fig. 1. Example of Backward Recovery

In a *service class*, the functional equivalence is defined according the WSs input and output attributes. A WS s is a functional substitute, denoted by $\equiv_F$, to another WS s^*, if s^* can be invoked with at most the input attributes of s and s^* produces at least the same output attributes produced by s. s is an Exact Functional substitute of s^*, denoted by $\equiv_{EF}$, if they have the same input and output attributes. Figure 2 illustrates several examples: $ws_1 \equiv_F ws_2$, however $ws_2 \not\equiv_F ws_1$, because ws_1 does not produce output a_5 as ws_2 does. $ws_1 \equiv_F ws_3$, $ws_3 \equiv_F ws_1$, and also $ws_1 \equiv_{EF} ws_3$.

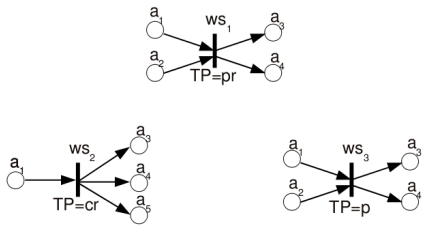


Fig. 2. Example of functional substitute WSs

In order to guarantee the TCWS global TP, a WS s can be replaced by another WS s^*, if s^* can behave as s in the recovery process. Hence, if $TP(s)$=p, in which case s only allows backward recovery, it can be replaced by any other

WS because all transactional properties allow backward recovery. A WS with $TP(s) = pr$ can be replaced by any other **retriable** WS (pr,$\boldsymbol{a}r$,cr), because all of them allow forward recovery. An **atomic** WS allows only backward recovery, then it can be replaced by any other WS which provides backward recovery. A **compensatable** WS can be replaced by a WS that also provides compensation as c and cr WSs. A cr WS can be only replaced by another cr WS because it is the only one allowing forward and backward recovery. Thus, a WS s is Transactional substitute of another WS s^*, denoted by $\equiv_T$, if s is a Functional substitute of s^* and their transactional properties allow the replacement.

In Figure 2, $ws_1 \equiv_T ws_2$, because $ws_1 \equiv_F ws_2$ and $TP(ws_2) = cr$, then ws_2 can behave as a pr WS; however $ws_1 \not\equiv_T ws_3$, even $ws_1 \equiv_F ws_3$, because as $TP(ws_3) = p$, w_3 cannot behave as a pr WS. Transactional substitution definition allows WSs substitution in case of failures.

When a substitution occurs, the faulty WS s is removed from the CPN-$TCWS_Q$, the new s^* is added, but we keep the original sequential relation defined by the input and output attributes of s. In that way, the CPN-$TCWS_Q$ structure, in terms of sequential and parallel WSs, is not changed. For **compensatable** WSs, it is necessary Exact Functional Substitute to do not change the compensation control flow in the respective BRCPN-$TCWS_Q$. In fact, when a **compensatable** WS is replaced, the corresponding compensate WS must be also replaced by the new corresponding one in the BRCPN-$TCWS_Q$. The idea is to try to finish the TCWS execution with the same properties of the original one.

2.4 Protocol in case of Failures

In case of failure of a WS s, depending on the $TP(s)$, the following actions could be executed:

- if $TP(s)$ is **retriable** (pr, $\boldsymbol{a}r$, cr), s is re-invoked until it successfully finish (forward recovery);
- otherwise, another Transactional substitute WS, s^*, is selected to replace s and the unrolling algorithm goes on (trying a forward recovery);
- if there not exist any substitute s^*, a backward recovery is needed, i.e., all executed WSs must be compensated in the inverse order they were executed; for parallel executed WSs, the order does not matter.

When in a *service class* there exist several WSs candidates for replacing a faulty s, it is selected the one with the best quality measure. The quality of a transition depends on the user query Q and on its QoS values. WSs Substitution is done such that the substitute WS locally optimize the QoS. If several transitions have the same value of quality, they can be randomly selected to be the substitute. A similar quality measure is used in [7] to guide the composition process. Then, during the execution, we keep the same heuristic to select substitutes.

3 FaCETa: An TCWS Executer with Backward and Forward Recovery Support

In this Section we present the overall architecture of FaCETa, our execution framework. The execution of a TCWS in FaCETa is managed by an Execu-

tion Engine and a collection of software components called Engine Threads, organized in a three levels architecture. In the first level the Execution Engine receives the TCWS (represented by a CPN). It is in charge of initiating, controlling, and monitoring the execution of the TCWS. To do so, it launches, in the second layer, an Engine Thread for each WS in TCWS. Each Engine Thread is responsible for the execution control of its WS. They receive WS inputs, invoke the respective WS, and forward its results to its peers to continue the execution flow. In case of failure, all of them participate in the backward or forward recovery process. Actual WSs are in the third layer. Figure 3 roughly depicts the overall architecture.

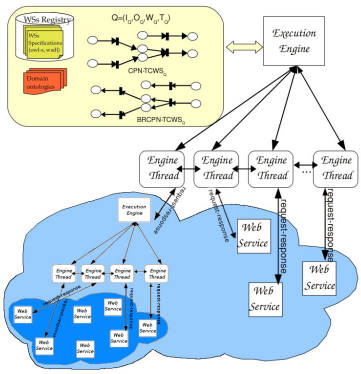
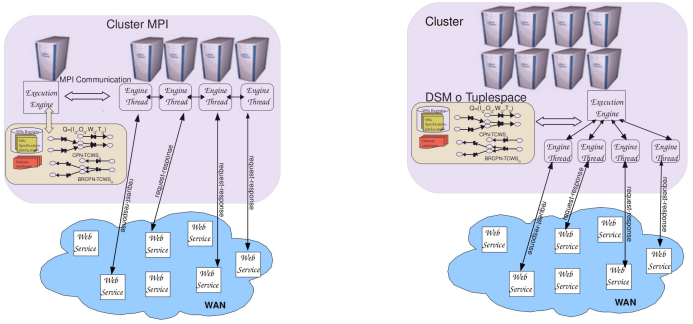


Fig. 3. FACETA Architecture

By distributing the responsibility of executing a TCWS across several Engine Threads, the logical model of our Executer allows distributed execution of a TCWS and is independent of its implementation, i.e., this model can be implemented in a distributed memory environment supported by message passing (see Figure 4(a)) or in a shared memory platform, e.g., supported by a distributed shared memory [22] or tuplespace [19] systems (see Figure 4(b)). The idea is to place the Executer in different physical nodes (e.g., a high available and reliable cluster computing) from those where actual WSs are placed. The Execution Engine needs to have access to the WSs Registry, which contains the *WSDL and OWLS* documents. The knowledge required at runtime by each Engine Thread (e.g., WS semantic and ontological descriptions, WSs predecessors and successors,transactional property, and execution control flow) can be directly extracted from the CPNs in a shared memory implementation or sent by the Execution Engine in a distributed memory implementation. In this paper, we have implemented a prototype of FACETA in a distributed memory platform using MPI.

Typically, a component of a TCWS can be a simple transactional WS or TCWS. Thus, we consider that transitions in the CPN, representing the TCWS,



(a) Distributed memory system (b) Distributed shared memory system

Fig. 4. Implementation of FaCETa.

could be WSs or TCWSs. WSs have its corresponding *WSDL and OWLS* documents. TCWSs can be encapsulated into an Executer; in this case the Execution Engine has its corresponding *WSDL and OWLS* documents. Hence, TCWSs may themselves become a WS, making TCWS execution a recursive operation (see Figure 3).

3.1 Distributed Memory Implementation of FaCETa

We implemented FaCETa in a MPI Cluster of PCs (i.e., a distributed memory platform) following a Master/Slaves-SPDM (Single Process Multiple Data) parallel model. The Execution Engine run in the front-end of the Cluster waiting user execution requests. To manage multiple client requests, the Execution Engine is multithreading. The deployment of a TCWS implies several steps: *Initial,* WS *Invocation*, and *Final* phases. In case of failures, recovery phases could be executed: *Replacing* phase, allowing forward recovery or *Compensation* phase for backward recovery.

Whenever the Execution Engine (master) receives a CPN-$TCWS_Q$ and its corresponding BRCPN-$TCWS_Q$, it performs the *Initial* phase: *(i)* start, in different nodes of the cluster, an Engine Thread (peer slaves) responsible for each transition in CPN-$TCWS_Q$, sending to each one its predecessor and successor transitions as CPN-$TCWS_Q$ indicates (for BRCPN-$TCWS_Q$ the relation is inverse) and the corresponding *WSDL and OWLS* documents (they describe the WS in terms of its inputs and outputs, its functional substitute WSs, and who is the compensation WS, if it is necessary); and *(ii)* send values of attributes in I_Q to Engine Threads representing WSs who receive them. Then the master wait for a successfully execution or for a message *compensate* in case of a backward recovery is needed.

Once Engine Threads are started, they receive the part of CPN-$TCWS_Q$ and BRCPN-$TCWS_Q$ that each Engine Thread concerns on, sent by the Execution Engine in the *Initial* phase. Then, they wait for the input values needed to invoke its corresponding WS. When an Engine Thread receives all input

values (sent by the master or by other peers) and all its predecessor peers have finished, it executes the WS *Invocation* phase, in which the actual WS is remotely invoked. If the WS finishes successfully, the ENGINE THREAD sends WS output values to ENGINE THREADS representing its successors and wait for a *finish* or *compensate* message. If the WS fails during the execution, the ENGINE THREAD tries a forward recovery: if TP(WS) is **retriable**, the WS is re-invoked until it successfully finish; otherwise the ENGINE THREAD executes the *Replacing* phase: the ENGINE THREAD has to determine the best substitute among the set of functional substitute WSs; it calculates the quality of all candidates according their QoS criteria values and the preferences provided in the user query; the one with the best quality is selected to replace the faulty WS; this phase can be executed for a maximum number of times ($MAXTries$). If replacing is not possible, the *Compensation* phase has to be executed: the ENGINE THREAD responsible of the faulty WS sends the message *compensate* to EXECUTION ENGINE and control tokens to successor peers of the compensation WS, in order to inform about this failure and start the unrolling process over BRCPN-$TCWS_Q$; once an ENGINE THREAD receives all control tokens, it invokes the compensation WS; the unrolling of BRCPN-$TCWS_Q$ ensure the invocation of compensation WSs, s', in the inverse order in which their corresponding WS, s, were executed. Note that forward recovery is executed only by the ENGINE THREAD responsible of the faulty WS, without intervention of the master neither other peers; while backward recovery need the intervention of all of them.

If the TCWS was successfully executed, in the *Final* phase the master receives all values of attributes of O_Q, in which case it broadcasts a *finish* message to all slaves to terminate them, and returns the answer to user; otherwise it receive a *compensate* message indicating that a backward recovery has to be executed, as it was explained above, and return an error message to user.

Assumptions: In order to guarantee the correct execution of our algorithms, the following assumptions are made: *(i)* the network ensures that all packages are sent and received correctly; *(ii)* the EXECUTION ENGINE and ENGINE THREADS run in a reliable cluster, they do not fail; *(iii)* the ENGINE THREADS receive all WS outputs when its corresponding WS finishes, they cannot receive partial outputs from its WS; and *(iv)* the component WSs can suffer silent or stop failures (WSs do not response because they are not available or a crash occurred in the platform); we do not consider runtime failures caused by error in inputs attributes (e.g., bad format or out of valid range) and byzantine faults (the WS still responds to invocation but in a wrong way).

4 Results

We developed a prototype of FACETA, using Java 6 and MPJ Express 0.38 library to allow the execution in distributed memory environments. We deployed FACETA in a cluster of PCs: one node for the EXECUTION ENGINE and one node for each ENGINE THREAD needed to execute the TCWS. 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 generated 10 **compensatable** TCWSs. All those TCWSs were automatically generated by a composition process [7], from synthetic datasets comprised by 800 WSs with 7 replicas each, for a total of 6400 WSs. Each WS is annotated with a transactional property and a set of *QoS* parameters, however for our experiments we only consider the response time as the *QoS* criteria. Replicas of WSs have different response times.

The OWLS-API 3.0 was used to parse the WS definitions and to deal with the OWL classification process.

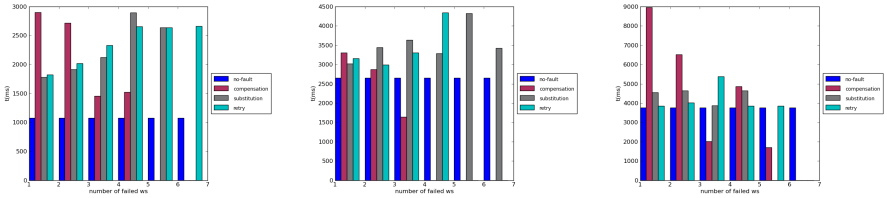
The first group of experiments were focussed on a comparative analysis of the recovery techniques. The second group of experiments evaluates the overhead incurred by our framework in control operations to perform the execution of a TCWS and to execute the fault tolerant mechanisms.

To simulate unreliable environments, we define five different conditions wherein all WSs have the same probability of failure: 0.2, 0.15, 0.1, 0.005, and 0.001. The executions on these unreliable environments were done in three scenarios to support the fails: *(i)* backward recovery (compensation, red bars in Figure 5), *(ii)* forward recovery because all WSs are retriable (retry, light blue bars in Figure 5), and *(iii)* forward recovery (substitution, gray bars in Figure 5). On each scenario all TCWSs were executed 10 times.

Each TCWS was also executed 10 times in a reliable environment, in which all WSs have 0 as probability of failures (no-faulty, blue bars in Figure 5) in order to classify them according their average total execution time in three groups: less than 1500ms (Figure 5(a)), (ii) between 1500ms and 3500ms (Figure 5(b), and (more than 3500ms (Figure 5(c)).

In Figure 5 we plot the average of the total execution time according the number of failed WSs, in order to compare all recovery techniques. The results show that when the number of failed WSs is small (i.e., the probability of failures is less than 20%) backward recovery (compensation) is the worst strategy because almost all component WSs had been executed and have to be compensated. Moreover, when the average of the total execution time of TCWSs is high (greater than 1500ms) forward recovery with retry strategy is better than forward recovery with substitution due to the substitute normally has a bigger response time than the faulty WS. By the other side, in cases in which the probability of failure is greater than 30%, backward recovery whit compensation behaves better than the other ones (even the final results is not produced) because there are many faulty services and only few have to be compensated.

Another issue that can be observed it is the number of outputs received before the backward recovery mechanism has to be executed. In this experiment, the average percentage of outputs received before compensation was 37%. All these outputs are lost or delivered as a set of incomplete (and possibly meaningless and useless) outputs to the user. This percentage is related to the average percentage of compensated services, which is 80%, confirming the overhead, the possible unfulfillment of *QoS* requirements, and the lost outputs. Definitely, backward



(a) Total Exec. Time less than 1500ms

(b) Total Exec. Time between 1500ms and 3500ms

(c) Total Exec. Time more than 3500ms

Fig. 5. Executions on the unreliable environments

recovery should be executed only in absence of another alternative, at early stages of execution of a TCWS, or high faulty environments.

To measure the intrusiveness of FACETA incurred by control activities, we execute the same set of experiments describe above, but we set to 0 the response time of all WSs. Table 4 shows the average overhead under all different scenarios.

	Average Overhead (ms)	% overhead increased
No Fault	611.7	
Compensation	622.38	2%
Substitution	612.82	0.2%
Retry	612.01	0.05%

Table 1. Average overhead incurred by FACETA

The average overhead of FACETA only depends on the number of components WSs in a TCWS. It does not depend on the total response time of TCWS. It means that while the total response time is higher the overhead % will decrease. It is clear that the reason behind the backward recovery strategy overhead (increased by 2%) is the amount of coordination required to start the compensation and the fact that a new WS execution (the compensation WS execution) has to be performed for each successfully executed WS, in order to restore the consistent system state. Additionally, the compensation process has to be done following the unrolling algorithm of the respective BRCPN-$TCWS_Q$. We do not consider to wait before the retry of a failed WS execution; therefore, the increased overhead of retry a WS is almost imperceptible.

As the *service class* for each WS is sent by the EXECUTION ENGINE in the *Initial* phase, each ENGINE THREAD has the list of the functional substitute WSs sorted according their quality, then there is virtually no overhead when searching for a functional substitute WS to replace a faulty one.

Based on the results presented above, we can conclude that FACETA efficiently implements fault tolerant strategies for the execution of TCWSs with admissible small overhead.

5 Related Work

Regarding fault tolerant execution of Composite WSs (CWSs), there exist centralized and distributed approaches. Generally centralized approaches [17, 23, 25] consider, besides compensation process, alternative WSs in case of failures or absent WSs, however they extend the classical 2PC protocol, which is time consuming, and they are not transparent to users and developers.

In distributed approaches, the execution process proceeds with collaboration of several entities. We can distinct two kinds of distributed coordination approach. In the first one, nodes interact directly based on a peer-to-peer application architecture and collaborate, in order to execute a CWS with every node executing a part of it [2, 5, 16, 18, 28]. In the second one, they use a shared space for coordination [4, 12, 19].

FENECIA framework [16] 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 [18], 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 TCWS back to a consistent termination state according to the termination protocol (by considering alternative services, replacements, and compensation). In [28] a fault handling and recovery process based on continuation-passing messaging, is presented. Nodes interpret such messages and conduct the execution of services without consulting a centralized engine. However, this coordination mechanism implies a tight coupling of services in terms of spatial and temporal composition. Nodes need to know explicitly which other nodes they will potentially interact with, and when, to be active at the same time. In [2] all replicas of a WS are simultaneously invoked. Only results of the first replica finished are accepted, other executions are halted or ignored. As our work, in [5] a rollback workflow is automatically created considering the service dependencies. Those frameworks support users and developers to construct CWS based on WS-BPEL technologies, then they are not transparent to users and developers.

Another series of works rely on a shared space to exchange information between nodes of a decentralized architecture, more specifically called a tuple space [12, 19]. The notion of a tuplespace is a piece of memory shared by all interacting parties. Using tuplespace for coordination, the execution of a (part of a) workflow within each node is triggered when tuples, matching the tem-

plates registered by the respective nodes, are present in the tuplespace. Thus, the templates a component uses to consume tuples, together with the tuples it produces, represent its coordination logic. In [19] approach to replace a centralized BPEL engine by a set of distributed, loosely coupled, cooperating nodes, is presented. This approach presents a coordination mechanism where the data is managed using a tuplespace and the control is driven by asynchronous messages exchanged between nodes. This message exchange pattern for the control is derived from a Petri Net model of the workflow. In [19], the workflow definition is transformed into a set of activities, that are distributed by passing tokens in the Petri Net. In [12] an alternative approach is presented, based on the chemical analogy. Molecules (data) are floating in a chemical solution, and react according to reaction rules (program) to produce new molecules (resulting data). The proposed architecture is composed by nodes communicating through a shared space containing both control and data flows, called the multiset. Through a shared multiset, containing the information on both data and control dependencies needed for coordination, chemical WSs are co-responsible for carrying out the execution of a workflow in the CWS in which they appear. Their coordination is decentralized and distributed among individual WS chemical engine executing a part of the workflow. As this approach, in our approach the coordination mechanism stores both control and data information independent of its implementation (distributed or shared memory). However, none of these works manage failures during the execution.

Facing our approach against all these works, we overcome them because the execution control is distributed and independent of the implementation (it can be implemented in distributed or shared memory platforms), it efficiently executes TCWSs by invoking parallel WSs according the execution order specified by the `CPN`, and it is totally transparent to users and WS developers, i.e., user only provides its TCWS, that could be automatically generated by the composition process [7] and no instrumentation/modification/specification is needed for WSs participating in the TCWS; while most of these works are based on WS-BPEL and/or some control is sitting closely to WSs and have to be managed by programmers.

There exist some recent works related to compensation mechanism of CWSs based on Petri-Net formalism [21, 24, 26]. The compensation process is represented by Paired Petri-Nets demanding that all component WSs have to be compensatable. Our approach considers other transactional properties (e.g., *pr*, *cr*, ***a****r*) that also allows forward recovery and the compensation Petri-Net can model only the part of the TCWS that is compensable. Besides, in those works, the Petri-Nets are manually generated and need to be verified, while in our approach they are automatically generated.

6 Conclusions and Future Work

In this paper we have presented FACETA, a framework for ensuring *correct and fault tolerant execution order* of TCWSs. The execution model is distributed, can be implemented in distributed or share memory systems, is independent of

implementation of WS providers, and is transparent to users and developers. To support failures, FACETA implements forward recovery by replacing the faulty WS and backward recovery based on a unrolling process over a CPN representing the compensation flow. We have presented a distributed memory implementation of FACETA in order to compare the behavior of both recovery techniques. The results show that FACETA efficiently implements fault tolerant strategies for the execution of TCWSs with small overhead.

We are currently working on implementing FACETA in a distributed shared memory platform in order to test the performance of the framework in centralized and decentralized platforms. Our intention is to compare both implementations under different scenarios (different characterizations of CPNs) and measure the impact of compensation and substitution on *QoS*.

References

1. Valdino Azevedo, Marta Mattoso, and Paulo Pires, *Handling dissimilarities of autonomous and equivalent web services*, Proc. of Caise-WES, 2003.
2. Johannes Behl, Tobias Distler, Florian Heisig, Rudiger Kapitza, and Matthias Schunter, *Providing fault-tolerant execution of web-service–based workflows within clouds*, Proc. of the 2nd Int. Workshop on Cloud Computing Platforms (CloudCP), 2012.
3. Antonio Brogi, Sara Corfini, and Razvan Popescu, *Semantics-based composition-oriented discovery of web services*, ACM Trans. Internet Techn. **8** (2008), no. 4, 1–39.
4. Paul Buhler and José M. Vidal, *Enacting BPEL4WS specified workflows with multiagent systems*, The Workshop on Web Services and Agent-Based Eng., 2004.
5. Omid Bushehrian, Salman Zare, and Navid Keihani Rad, *A workflow-based failure recovery in web services composition*, Journal of Software Engineering and Applications **5** (2012), 89–95.
6. Yudith Cardinale, Joyce El Haddad, Maude Manouvrier, and Marta Rukoz, *Web service selection for transactional composition*, Elsevier Science-Procedia Computer Science Series (Int. Conf. on Computational Science(ICCS) **1** (2010), no. 1, 2689–2698.
7. Yudith Cardinale, Joyce El Haddad, Maude Manouvrier, and Marta Rukoz, *CPN-TWS: A colored petri-net approach for transactional-qos driven web service composition*, Int. Journal of Web and Grid Services **7** (2011), no. 1, 91–115.
8. Yudith Cardinale, Joyce El Haddad, Maude Manouvrier, and Marta Rukoz, *Transactional-aware Web Service Composition: A Survey*, IGI Global - Advances in Knowledge Management (AKM) Book Series, 2011.
9. Yudith Cardinale and Marta Rukoz, *Fault tolerant execution of transactional composite web services: An approach*, Proc. of The Fifth Int. Conf. on Mobile Ubiquitous Computing, Systems, Services qnd Technologies (UBICOMM), 2011.
10. Yudith Cardinale and Marta Rukoz, *A framework for reliable execution of transactional composite web services*, Proc. of The Int. Conf. on Management of Emergent Digital EcoSystems (MEDES), 2011.
11. Joyce El Haddad, Maude Manouvrier, and Marta Rukoz, *TQoS: Transactional and QoS-aware selection algorithm for automatic Web service composition*, IEEE Trans. on Services Computing **3** (2010), no. 1, 73–85.

12. Hector Fernandez, Thierry Priol, and Cédric Tedeschi, *Decentralized approach for execution of composite web services using the chemical paradigm*, IEEE Int. Conf. on Web Services, 2010, pp. 139–146.
13. Walid Gaaloul, Sami Bhiri, and Mohsen Rouached, *Event-based design and runtime verification of composite service transactional behavior*, IEEE Trans. on Services Computing **3** (2010), no. 1, 32–45.
14. Chad Hogg, Ugur Kuter, and Hector Munoz-Avila, *Learning Hierarchical Task Networks for Nondeterministic Planning Domains*, The 21st Int. Joint Conf. on Artificial Intelligence (IJCAI-09), 2009.
15. Farrell J. and H Lausen, *Semantic annotations for wsdl and xml schema*, January 2007, W3C Candidate Recommendation. http://www.w3.org/TR/sawsdl/.
16. Neila Ben Lakhal, Takashi Kobayashi, and Haruo Yokota, *FENECIA: failure endurable nested-transaction based execution of compo site Web services with incorporated state analysis*, VLDB Journal **18** (2009), no. 1, 1–56.
17. An Liu, Liusheng Huang, Qing Li, and Mingjun Xiao, *Fault-tolerant orchestration of transactional web services*, Web Information Systems – WISE 2006 (Karl Aberer, Zhiyong Peng, Elke Rundensteiner, Y anchun Zhang, and Xuhui Li, eds.), Lecture Notes in Computer Science, vol. 4255, Springer Berlin-Heidelberg, 2006, pp. 90–101.
18. An Liu, Qing Li, Liusheng Huang, and Mingjun Xiao, *FACTS: A Framework for Fault Tolerant Composition of Transactional Web Services*, IEEE Trans. on Services Computing **3** (2010), no. 1, 46–59.
19. Daniel Martin, Daniel Wutke, and Frank Leymann, *Tuplespace middleware for petri net-based workflow execution*, Int. J. Web Grid Serv. **6** (2010), 35–57.
20. S. McIlraith, T.C. Son, and H. H. Zeng, *Semantic web services*, IEEE Intelligent Systems **16** (2001), no. 2, 46–53.
21. Xiaoyong Mei, Aijun Jiang, Shixian Li, Changqin Huang, Xiaolin Zheng, and Yiyan Fan, *A compensation paired net-based refinement method for web services composition*, Advances in Information Sciences and Service Sciences **3** (2011), no. 4.
22. Jesús De Oliveira, Yudith Cardinale, Jesús Federico, Rafael Chacón, and David Zaragoza, *Efficient distributed shared memory on a single system image operating system*, Latin-American Conf. on High Performance Computing, 2010, pp. 1–7.
23. Jonghun Park, *A high performance backoff protocol for fast execution of composite web services*, Computers and Industrial Engineering **51** (2006), 14–25.
24. Fazle Rabbi, Hao Wang, and Wendy MacCaull, *Compensable workflow nets*, Formal Methods and Software Engineering - 12th Int. Conf. on Formal Engineering Methods, LNCS, vol. 6447, 2010, pp. 122–137.
25. M. Schafer, P. Dolog, and W. Nejdl, *An environment for flexible advanced compensations of web service transactions*, ACM Transactions on the Web **2** (2008).
26. Yonglin Wang, Yiyan Fan, and Aijun Jiang;, *A paired-net based compensation mechanism for verifying Web composition transactions*, The 4th Int. Conf. on New Trends in Information Science and Service Science, 2010.
27. Qi Yu, Xumin Liu, Athman Bouguettaya, and Brahim Medjahed, *Deploying and managing web services: issues, solutions, and directions*, The VLDB Journal **17** (2008), 537–572.
28. Weihai Yu, *Fault handling and recovery in decentralized services orchestration*, The 12th Int. Conf. on Information Integration and Web-based Applications #38; Services, iiWAS, ACM, 2010, pp. 98–105.