FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 38 (2013) No. 3

DOI: 10.2478/fcds-2013-0006

ISSN 0867-6356
e-ISSN 2300-3405

Applying Message Logging to Support Fault-Tolerance of SOA Systems

Arkadiusz Danilecki [*], Mateusz Hołenko [†], Anna Kobusińska [‡],
Michał Szychowiak [§], Piotr Zierhoffer [¶]

Abstract. This paper addresses a problem of increasing fault-tolerance of service-oriented systems built of RESTful web services. To solve such a problem, rollback-recovery protocol is proposed. The protocol employs known rollback-recovery techniques, however, it modifies and specially adjusts them for specific characteristics of the SOA systems. The paper includes a proof of safety property of the proposed protocol.

Keywords: SOA, fault-tolerance, rollback-recovery protocol, correctness

1 Introduction

In the recent years, the rapid growth of development and deployment of service-oriented systems (SOA) has been observed. The basic SOA assumption is to create business applications based on loosely-coupled, autonomous services, which are implemented by software modules that operate accordingly to the established criteria, and represent the specific functionality [11]. Such an approach allows to create new, complex applications, as well as to integrate the existing systems. In the consequence, in the SOA, an unprecedented so far flexibility in the design of distributed applications is achieved.

Although SOA-based applications have many advantages, they are also highly error-prone. Failures of the SOA components, lead to limitations in the availability of services, thus affecting the reliability of the whole system. Such a situation is highly undesirable from the viewpoint of SOA clients, who expect that provided services

[*] arkadiusz.danilecki@cs.put.poznan.pl, Poznań University of Technology
[†] mateusz.holenko@cs.put.poznan.pl, Poznań University of Technology
[‡] anna.kobusinska@cs.put.poznan.pl, Poznań University of Technology
[§] michal.szychowiak@cs.put.poznan.pl, Poznań University of Technology
[¶] piotr.zierhoffer@cs.put.poznan.pl, Poznań University of Technology

are reliable and available, and assume an uninterrupted business processing. Ensuring reliability of the SOA systems is particularly important, due to their practical applications in various domains, among which are tele-medicine systems, real-time traffic information and navigation systems, mobile Internet stock trading systems, and many others. In such systems the loss of data or the suspension of the system functionality is unacceptable, as it may lead to the financial loss of companies, result in the loss of life or contribute to lower company's reputation in the market. Therefore, design of mechanisms increasing the SOA systems' fault-tolerance, and enabling the consistent continuation of the processing despite failures, is a very important and current research objective, motivated by both a great market potential and by many challenging research problems.

There are many mechanisms increasing the reliability of general distributed systems proposed in the literature, among which one of the most important, and frequently used is the backward-recovery approach [4]. Unfortunately, the direct use of known rollback-recovery mechanisms in the context of SOA systems faces a number of problems arising from the SOA specificity. They include the autonomy of the services, dynamic nature of the interaction; longevity of interaction; and the inherent constant interaction with the outside world, among the others. For example, the classical solutions using the mechanisms of checkpointing [4] require either to control when checkpoints are taken, or the appropriate choice of the checkpoint used during the process recovery. In the case of the SOA systems such solutions cannot be applied, due of the autonomy of services, which is expressed among the others in the implementation of services' own fault-tolerant policies. In the result a service sometimes cannot be forced to take a checkpoint, or to rollback, as well as it may refuse to inform other services on checkpoints it has taken. Another limitation, which arises from the fact that every service invocation may result in irrevocable changes, is the necessity of applying so-called output-commit protocols [4], in which checkpoints are taken every time when the external interaction is performed. The assumed SOA model imposes also certain restrictions on the rollback-recovery of services. The failure of one service can not affect the availability of other services taking part in the processing. This means that the rollback-recovery of one service neither should cause the cascading rollback of other services, nor influence their state.

Consequently, the existing solutions have to be revised, and specially profiled for the SOA environments to efficiently meet their requirements, and to take advantage of their specifics. Therefore in this paper a rollback-recovery protocol for SOA systems based on the REST paradigm is proposed. The proposed protocol ensures that in the case of failure of one or more system components (i.e., web services or their clients), a coherent state of distributed processing is automatically recovered. While the protocol can be used in any SOA environment, it is particularly well-suited for processing, which does not have the transactional character, and where clients applications do not use the business process engines.

The paper is structured as follows. The related work is characterized in Section 2. System model and basic definitions are presented in Section 3. Section 4 and 5 describe general idea of the proposed rollback-recovery protocol, and present its implementation respectively. The proof of safety follows in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

To improve reliability of SOA-based systems, transaction processing, and a mechanism of compensations is commonly used [3]. The compensation of operations, is realized in the SOA as the invocation of compensation services. A limitation of this approach is the necessity of providing all compensation services in advance, and the proper integration of the compensation invocations into processing, to ensure that the intended purpose of the rollback has been actually achieved. Compensation mechanism can be also employed when transactions are rolled back for reasons not related to the failures of the system components (e.g., in the case of failures at the business logic level). Since the transactional approach [8] is burdened with high costs of maintenance transactions' properties, its use is not viable in applications that only require reliability.

Mechanisms improving reliability are to some extent implemented by many business processes engines (e.g., BPEL engines [7]). A common approach used by such engines is the forward recovery, mostly reduced to partially automatic retry of the failed operations. The use of BPEL engines, and mechanisms they offer, cannot solve all the problems related to the issues of ensuring system reliability. Existing solutions increase the reliability of only a single component, which is a local instance of a business process implemented by the engine, without taking into account the potential dependencies between a nested services. As a result, such engines do not guarantee the preservation of exactly-once semantics for non-idempotent requests, unless additional protocols are employed (such as WS-ReliableMessaging [9]). They also do not provide a fully automated and transparent recovery.

3 System Model

Throughout this paper a distributed service-oriented system is considered. The system consists of a number of autonomous, loosely-coupled RESTful web services [5, 11, 10], exposed as resources, and identified by a uniform resource identifiers upon which a fixed set of HTTP operations is applied. Thus, a client who wants to use a service communicates with it via a standardized interface, e.g., GET, PUT, POST and DELETE methods [10], and exchanges representations of resources. It is assumed that both, clients and services are piece-wise deterministic, i.e., they generate the same results (in particular, the same URIs for new resources) in the result of a multiple repetition of the same requests, assuming the same initial state. Services can concurrently process only clients' requests that do not require access to the same or interacting resources. Otherwise, the existence of a mechanism serializing access to resources, which uniquely determines the order of operations, is assumed.

The communication model used in the paper is based on a request-response approach, and does not guarantee the correct delivery of messages (they may be lost or duplicated). The considered communication channels do not provide FIFO property. Additionally, the crash-recovery model of failures is assumed, i.e., system components may fail and recover after crashing a finite number of times [1]. Failures may happen at arbitrary moments, and we require any such failure to be eventually detected, for example by a Failure Detection Service [2].

We assume that each service provider may have its own reliability policy, and may use different local mechanisms that provide fault tolerance. Therefore, in the paper, by a recovery point we denote an abstraction describing a consistent state of the service, which can be correctly reconstructed after a failure, but we do not make any assumptions on the how and when such a recovery points are made (to make a recovery point logs, checkpoints, replicas and other mechanisms may be used). It is assumed that each service takes recovery points independently (and has at least one recovery point, representing it's initial state). Similarly, the client may also provide its own fault tolerance techniques to save its state.

4 General Idea of Rollback-Recovery Protocol

Due to the fact that processing in the SOA is based on the processing of messages, and the communication in systems based on the REST paradigm is stateless (each message sent by a client does not depend on the message preceding it, and contains all the necessary information required by a service to perform it), it can be observed that if the business process participants are piece-wise deterministic, their states are reflected in the history of their communication. This means that after the failure of any client or service, the reprocessing of their messages in the same order as before the failure leads to the consistent system state (i.e., one that could be achieved during the failure-free processing). Based on this observation, we propose a protocol that adopts the well-known technique of message logging [4] to increase the reliability of processing in SOA systems.

The proposed protocol logs messages sent between business process participants during the failure-free processing, to be able in the case of failure of one or more system components to recover a consistent state of processing by resending saved messages in a specified order. However, since in the SOA participants of business processing may have their private mechanisms providing reliability, after the failure occurrence the state of processing participants may be partially reconstruct with the use of these local mechanisms. The recovery of a local state may include the reprocessing of some messages belonging to the history of global communication. Therefore, only those messages, the processing of which was not reflected in services' (clients') recovered state, should be processed again. Such messages have to be found and performed, which is the task of the proposed rollback-recovery protocol.

The proposed protocol introduces three types of components: Recovery Management Units (RMU), Client Intermediary Modules (CIM), and Service Intermediary Modules (SIM). RMU stores the requests and responses send among business process participants in the Stable Storage able to survive all failures. The saved history of communication is then used during rollback and recovery of processing system state. Since there are many RMUs, the history of communication is dispersed among

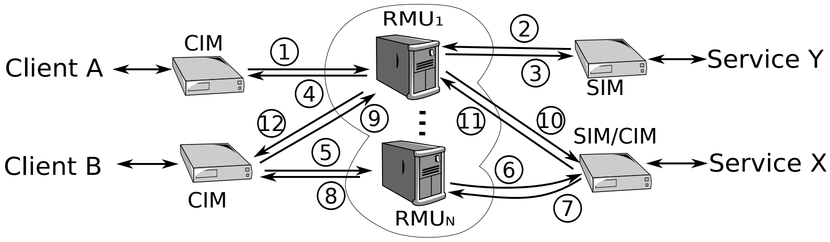

Figure 1: Diagram of failure-free processing

them. When the processing starts, each of its participants registers itself in the selected RMU. Each service is registered in one RMU, but the single RMU can be used by many services. In turn, the client can be registered simultaneously in many RMUs, but always one of them (called also a master RMU) stores information on other RMU's used by the client.

CIMs and SIMs hide the details of rollback-recovery protocol to clients and services respectively. For this purpose, both modules intercept messages issued by clients and servers, so they allow to fully control the flow of messages in the system. In fact, *CIM* and *SIM* serve as proxies for clients and servers. Additionally, SIMs monitor the service status and react in the case of its eventual failure by initiating and managing the service rollback-recovery procedure.

Fig. 1 presents the subsequent steps realised by the proposed rollback-recovery protocol. The request issued by a client to a chosen service is intercepted by the client's CIM, and forwarded to the client's master RMU. If the required service is registered in the RMU, the request is saved in the RMU's stable storage and then forwarded to the service through its SIM. Otherwise, client's master RMU obtains the URI of requested service RMU from its SIM, and sends back this information to the CIM, which reissues the request to a proper URI. The service performs request and sends the response back to RMU. The response is saved in the stable storage and forwarded to the client through its CIM. Fig. 1 illustrates the idea of the proposed protocol. Client A and services X, Y are registered in RMU_1, while client B is registered in RMU_N. The request submitted by client A to service Y follows steps (1) and (2), and after being proceeded the response is sent back to the client — steps (3) and (4). In turn, if client B submits the request to server X, it is received by RMU_N (5), the URI of $RMU_1$is obtained by RMU_N from SIM of requested service (6,7), and forwarded to CIM_B (8). Then, the request is resubmitted through RMU_1 (9). Further processing is carried out analogously — steps (10,11,12).

Actions taken in the case of client's or service's failure are presented in Fig. 2 and Fig. 3, respectively. It is assumed that due to the HATEOAS principle of Resource Oriented Architecture, in order to recover client's state, the last response obtained by the client before the failure occurrence should be resent. Since, in general the client communicates with many RMUs , such a last response should be chosen on the basis of information received from each of *RMUs*, the client contacted before its failure (the list of such RMUs is stored by the client's master RMU). The client's CIM

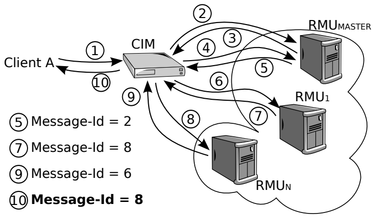

Figure 2: Client rollback-recovery

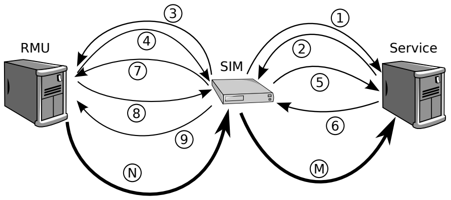

Figure 3: Service rollback-recovery

gathers informations from RMUs , and sends a response with the highest identifier to the client. The client then proceeds with the execution. To demonstrate the client's recovery let us consider Fig.2. The CIM of client A obtains a list of RMUs from RMU_{MASTER} (1). CIM requests all RMUs from the obtained list to send it the identifier of the last saved response sent to client A (2,3,4). When all RMUs respond (5,6,7), CIM selects message with the highest identifier and sends it to client.

When the service fails, its SIM starts the rollback-recovery process. First it finds the recovery point to which the service has to be rolled back (the one that contains only the responses not later, than the last one stored in the RMU), what is depicted by (1) in Fig. 3. The response contains the information on recovery points and identifiers of the last responses stored in each of them (2). Then SIM obtains from RMU the identifier of last response saved in its stable storage (3,4), and determines which recovery point contains messages not later then the last one saved in the RMU. The service state is rolled back to the chosen recovery point (5). When the rollback is finished (6), SIM requires RMU to resubmit all requests performed by the service before the failure, and not saved in the chosen recovery point (7). RMU acknowledges how many requests will be resubmitted (8). When SIM confirms that it's ready to begin the recovery process (9), all requests chosen in step (7) are sent again by the RMU (N) and re-executed by the service, in the same order as before the failure (M).

Beside failures of services and clients, also the components of infrastructure used by the rollback-recovery protocol can crash. Let us first consider the RMU's failure. During the failure-free work, RMU first records the obtained data in its *Stable Storage*, and afterwards it continues processing. All data vital to recovery (i.e., requests, responses and metadata) is recovered from the RMU's *Stable Storage*. Since it is highly undesirable to suspend business processes execution because of the RMU failure, the failure detection service is used to monitor state of RMU and to automatically start its recovery when the failure occurs. This service also notifies all services that RMU is restarted. Services registered in restarted RMU before its failure, have to register themselves again. Because the RMU failure causes all active connections to be closed, the registration process entails the rollback-recovery of services, in order to reissue all requests that might have been lost in the result of failure.

Failures of *SIM* and *CIM* do not affect the correctness of processing. All information required for the proper functioning of both modules are contained in their configuration files or received from the *RMU* during *SIM* and *CIM* initialization.

CIM ClientId :: *Client identifier*

int MsgId :: *Message identifier*

int IncId :: *Identifier of consecutive client's sessions*

int EpochId :: *Identifier of consecutive service's sessions*

SIM ServId :: *Service identifier*

URI ServURI,ResURI :: *URI of the service, resource*

int RespId :: *Order identifier of responses,given by the service*

set¡Request¿ SavedReqs :: *Set of saved requests*

set¡Response¿ SavedResps :: *Set of saved responses*

{Normal — Recovery} Mode :: *Current mode of SIM*

int SyncPts :: *Set of response id of repeated messages without group IDs*

int MsgToRsnd :: *Number of messages to be repeated during recovery*

Figure 4: Symbols used in rollback-recovery protocol

Therefore in the case of the failure of these modules no important data is lost. However, as it was in the case of *RMU*, all active connections processed before the failure are lost. Therefore on every restart of *SIM*, the service recovery process is started to recover the processing closed due to the error. *CIM* failure affects only clients that use it directly. In case of failure it is enough to restart *CIM* process.

Finally, let us consider the communication failures. The situation when the failure occurs before the client's request reaches *RMU* is indistinguishable (from the perspective of *RMU* and service) from the situation when no message was sent. In such case the client has to send a request again. Also the case, when communication failure occurs after *RMU* has processed the request does not affect the correctness and consistency of processing. *RMU* is able to determine if the request has been already processed and reject duplicate requests assuming that the response is saved in *Stable Storage*. In the result, the proposed protocol ensures that requests are processed exactly once by services. The problem, however, arises when the failure occurs during communication between *RMU* and a service. In such case it cannot be verified if the request has been obtained and processed by the service or not. Resending request in the first case would lead to an inconsistent state (where the request was processed twice). Therefore, to ensure that the final state of processing is always consistent, the worst case is assumed and request cannot be resend. Instead the service is rolled-back and recovered.

5 Protocol Implementation

Below, we describe the proposed rollback-recovery protocol, executed by *CIM*s, *SIM*s and *RMU*s. For the clarity of presentation, we omitted some details. In the presented pseudocodes we use the symbols presented in Fig. 4

Fig. 5 presents basic actions taken by the *CIM*. The request issued by the client is augmented by *CIM* with the required identifiers (l. 1-2). If *CIM* possesses in its

```
Upon receiving request $req$ of type Request from client $C_i$ at module $CIM_i$
 1: $req.ClientId \leftarrow CIM_i$
 2: $req.IncId \leftarrow IncId$
 3: if $\exists ce \in ServiceCache : ce.ServURI = req.ServURI$ then
 4:     send $req$ to $ce.RMU$
 5: else
 6:     send $req$ to $MasterRMU$
 7: end if

Upon receiving response $res$ of type Response from $RMU_k$ module to client $C_i$ at module $CIM_i$
 8: send $res$ to $C_i$

Upon receiving request $req$ of type GetLastResponse from client $C_i$ at module $CIM_i$
 9: foreach distinct $RMU \in \{ServiceCache \cup \{MasterRMU\}\}$ do
10:     send $req$ to $RMU$
11: end for
12: await all responses of type LastResponse
13: $MaxMsgId \leftarrow \mathbf{max}(r.MsgId : r \in ReceivedResps)$
14: send $(r : r \in ReceivedResps \wedge r.MsgId = MaxMsgId)$ to $C_i$
```

Figure 5: Rollback-recovery protocol - module CIM

cache the information on RMU of requested service, then it redirects the submitted request to such an RMU (l. 3-4).

Otherwise, the request is forwarded to the client's master RMU (l. 6). The response to request sent by client is obtained by CIM from RMU, and is transmitted to the client (l. 8).While performing a recovery of client's state, CIM asks all RMUs it cooperated with (l. 9-11) for the latest response sent to it (l. 9-13). From the set of obtained responses the one with the highest identifier is chosen (l. 14-15). Such a response is sent to the client (l.17).

In Fig. 6 actions taken by the RMU are discussed. First, the way in which RMU handles requests submitted by clients to services is described in lines 1-20.The RMU supports only requests that are submitted to services registered in it. If the requested service is registered in different RMU, the client is redirected to the proper RMU's URI. (l. 4-8). RMU performs only requests, which IncID (specifying the identifier of session to which the request belongs), corresponds to the current session identifier (l.1-3). If the response to request issued by the client has already been saved in the RMU, then there is no need to send this request to the service once again, as the already saved response can be sent to the client immediately (l. 9-14). With this solution, the same message (i.e., the message with the same identification number) may be sent by the client many times, without the danger of multiple service invocations. Hence, the idempotence of all requests is ensured. Otherwise, the request is saved in the *Stable Storage* and forwarded with the necessary information to the service (l.16-19). The response from the service to the client is suspended by RMU until all previous responses generated by the service (with lower RespId) are saved in the *Stable Storage* (l. 21-23). Then the considered response is saved in *Stable Storage* and forwarded to CIM (l.24-29). The exact steps taken during client's rollback-recovery are shown in

```
Upon receiving request $req$ of type Request from $CIM_i$ directed to $SIM_j$ at module $RMU_k$
if $Inc[CIM_i] \neq req.IncId$ then
    discard $req$ and exit
end if
if $SIM_j \notin RegServices$ then
    $r \leftarrow$ RMU of the service $SIM_j$
    $Redir[CIM_i] \leftarrow Redir[CIM_i] \cup \{r\}$
    send Redir $\langle r \rangle$ to $CIM_i$ and exit
end if
if $req \in SavedReqs$ then
    if $\exists\ r \in SavedResps : r.Req = req$ then
        send $r$ to $CIM_i$
    else
        send TryAgain to $CIM_i$
    end if
else
    $req.ServId \leftarrow SIM_j$
    $SavedReqs \leftarrow SavedReqs \cup \{req\}$
    $req.EpochId \leftarrow Epoch[SIM_j]$
    send $req$ to $ServiceURI$
end if
```

```
Upon receiving an error while waiting for response $res$ of type Response from $SIM_j$ directed to $CIM_i$ at module $RMU_k$
send StartRecovery to $SIM_j$
```

```
Upon receiving request $last$ of type GetLast from $SIM_j$ at module $RMU_k$
$LastRespId \leftarrow \mathbf{max}(resp.RespId : r \in SavedResps \wedge r.Req.ServId = SIM_j)$
send LastSaved $\langle LastRespId \rangle$
```

```
Upon receiving response $res$ of type Response for request $req$ of type Request from $SIM_j$ at module $RMU_k$
if $res.RespId \neq$ null then
    await $res.RespId = \mathbf{max}(r.RespId : r \in SavedResps \wedge r.Req.ServId = SIM_j) + 1$
end if
$res.Req \leftarrow req$
$SavedResps \leftarrow SavedResps \cup \{res\}$
if $req$ was not send in recovery process and is directed to $CIM_i$ then
    send $res$ to $CIM_i$
end if
```

```
Upon receiving request $resend$ of type ResendMsgs $\langle LowestRespId \rangle$ from $SIM_j$ at module $RMU_k$
$rec \leftarrow$ new StartRecovery
$Epoch[SIM_j] \leftarrow Epoch[SIM_j] + 1$
$rec.EpochId \leftarrow Epoch[SIM_j]$
$ReqsToRep \leftarrow \{r.Req : r \in SavedResps \wedge r.Req.ServId = SIM_j \wedge r.RespId \geq resend.LowestRespId\} \cup \{req \in SavedReqs : req.ServId = SIM_j \wedge (\nexists rsp \in SavedResps : rsp.Req = req)\}$
$rec.MsgToRsnd \leftarrow |r \in ReqsToRep : \exists rsp \in SavedResps \wedge rsp.Req = r|$
$rec.SyncPts \leftarrow \{req.RespId : req \in ReqsToRep \wedge req.GroupId = \mathbf{null}\}$
map¡Name, Value¿ $G$
foreach $req \in ReqsToRep$ do
    foreach $grp \in req.GroupId$ do
        if $(grp.Name \notin G.Names) \vee G[grp.Name] > grp.Value$ then
            $G[grp.Name] \leftarrow grp.Value$
        end if
    end for
end for
$rec.Groups \leftarrow G$
send $rec$ to $SIM_j$
foreach $req \in ReqsToRep$ do
    $req.EpochId \leftarrow Epoch[SIM_j]$
    send $req$ to $SIM_j$
end for
```

```
Upon receiving request $req$ of type GetLastResp from client $CIM_i$ at module $RMU_k$
$last \leftarrow$ new LastResp
$last.MsgId \leftarrow$ minimal integer value
foreach $r \in SavedResp : r.Req.ClientId = CIM_i$ do
    if $r.Req.MsgId > last.MsgId$ then
        $last.MsgId \leftarrow r.Req.MsgId$
        $last.OrignalResp \leftarrow r$
    end if
end for
send $last$ to $CIM_i$
```

Figure 6: Rollback-recovery protocol - module RMU

lines 31-39. If client contacted with many *RMU*s, they all take part in its recovery and inform the client's *CIM* about the last response sent to the client, before the failure occurrence (l. 40-41).

In turn, while the service recovery *SIM* gets from *RMU* the information on the last response obtained from the given service, and on this basis it distinguishes the set of requests that have to be resubmitted to the service (l. 42-61). Finally, in a Figure 7 the actions taken by *SIM* are presented. *SIM* works in two modes: normal and recovery. Requests obtained in the normal mode, after verification of the EpochId, are immediately delivered to the service (l. 21-25). In turn, in the recovery mode (l. 27-38) first the request type is checked. Normal requests (i.e., requests that are not resend by the *RMU*) are suspended until the end of the recovery process (l. 21-26). Resent requests are queued according to their RespId and suspended until all previous requests are not processed by the service. When all resent requests are processed *SIM* returns to the normal mode (l. 40).

6 Safety of the protocol

Lemma 1. *All requests received by the service S and responses received from services by the client C are stored in the Stable Storage.*

Proof. Let us consider a request req submitted by client C_i to service S_j, and response res returned by the service to the client. From the algorithm, all messages sent between services and clients are intercepted by their CIM and SIM modules, and are forwarded to RMU (Fig. 5, l. 4,6 and Fig. 7, l. 4). RMU, before sending req to SIM of S, and before forwarding res to CIM of C, saves them in the stable storage (Fig. 6, l. 19, 28). □

Lemma 2. *All requests submitted by clients to service S, and performed by S before a failure, are reflected in the recovered service state.*

Proof. In the result of assumption on service determinism and isolation of requests, the recovery points made by a service accordingly to its local fault-tolerant policy, correspond to consistent service states that actually occurred during the failure-free processing of requests. Therefore, such recovery points may be used as an initial states while applying rollback-recovery.

Accordingly to the protocol, SIM and RMU negotiate to which recovery point the service will be rolled back, and which requests were performed after the chosen recovery point and thus have to be resubmitted (Fig. 6. l. 22-23, 35, and Fig. 7, l. 6-17). Since every request obtained by the service before the failure is also saved in the RMU (Lemma 1), all requests from the negotiated set may be resubmitted by the RMU (Fig. 6 , l. 48-50). Due to the fact, that the order of resubmitted requests corresponds to the order in which requests were performed before the failure (Fig. 7, l. 27-36), the recovered service state is the same as the state before the failure. The recovered client state is consistent with the state of client occurring in the failure-free processing. The recovery of the client differs depending on the client's requirements. For some clients, the last response from the service may be enough for recovery.

```
Upon receiving request $req$
of type Request from $RMU$
while $Mode = Normal$ at $SIM_j$
if req.EpochId = EpochId then
    send $req$ to $Service$
    await $resp$ of type Response
    send $resp$ to $RMU$
end if

procedure StartRecoveryProcess
send GetCheckpoints to $Service$
await $ckpt$ of type Checkpoints
send GetLast to $RMU$
await $last$ of type LastSaved
$RbCkpt \leftarrow Ckpt_i \in ckpt.Ckpts : Ckpt_i.LastRespId \leq last.RespId \wedge \forall_{j>i} Ckpt_j.LastRespId > last.RespId$
send to $Service$ Rollback $\langle RbCkpt.CkptId \rangle$
send to $RMU$ ResendMsgs $\langle RbCkpt.LastRespId + 1 \rangle$
await $recov$ of type StartRecovery
$MsgToRsnd \leftarrow recov.MsgToRsnd$
$SyncPts \leftarrow recov.SyncPoints$
$EpochId \leftarrow recov.EpochId$
$CurGroups \leftarrow recov.Groups$
end procedure

Upon receiving request $req$
of type Request from $RMU$
while $Mode = Recovery$
if $req.EpochId \neq EpochId$ then
    discard $req$ and exit
end if
if $req.RespId$ = null then
    wait until $Mode = Normal$
    send $req$ to $Service$
    await $resp$ of type Response
    send $resp$ to $RMU$
else
    wait until $(\nexists\ sync \in SyncPts : sync < req.RespId) \wedge \forall_{g \in req.GroupId} (CurGroups[g.Name] \neq$ null $\wedge\ g.Value = CurGroups[g.Name] + 1)$
    send $req$ to $Service$
    await $resp$ of type Response
    if $\exists\ p \in SyncPts\colon\ p = res.RespId$ then
        $SyncPts \leftarrow SyncPts \setminus \{p\}$
    else
        foreach $g \in resp.GroupId$ do
            $CurGroups[g.Name] \leftarrow g.Value$
        end for
    end if
    $MsgToRsnd \leftarrow MsgToRsnd - 1$
    send $resp$ to $RMU$
end if

When $MsgToRsnd = 0$
$Mode \leftarrow Normal$
```

Figure 7: Rollback-recovery protocol - module SIM

According to the protocol, such clients contact the RMU, get the last response they obtained before the failure and then directly proceed with the execution (Fig. 5 l. 12-14, Fig. 6, l. 52-60). If the last response is not sufficient for client's recovery, the client first recovers using its own local checkpoints or logs. Next the client proceeds with processing, sending requests to the CIM, which then forwards them to the RMU (Fig. 5, l. 1-7), like during the failure-free processing. If the RMU already has the response for the request, such a response is sent to the client (Fig. 6, l. 9-11). Since the client is piece-wise deterministic, its state is reconstructed up to the point of the last request sent before the failure. □

Lemma 3. *Requests resubmitted by a client to a service do not lead to the inconsistent service state.*

Proof. By assumption, if the resubmitted client's request is obtained by RMU, which had already processed such a request (the request with the same identifier), the request is not handled by the RMU (Fig. 6, l. 9-14). Therefore, such a request will not be sent to the service, and thus will not influence the service's state. □

Theorem 1. *The proposed rollback-recovery protocol for service-oriented systems provides, in the case of failure of one or more system components, a recovery of a consistent system state.*

Proof. The system state consists of states of its individual elements: clients, services, RMUs, CIMs and SIMs. According to Lemma 2, the recovered service state, and its state before the failure are indistinguishable. Since when failures do not occur, the service's state is consistent, thus also the recovered state is consistent. Moreover, the rollback and recovery of one service never requires rollback of any other service, since the rollback-recovery protocol recovers services' states independently from each other, so the rollback-recovery of a service will not influence other services states.

According to the Lemma 2, clients also possess mechanisms to correctly recover their states. Moreover, as stated by Lemma 3, the rollback-recovery of client's processing does not lead to inconsistency on the service side.

Due to the fact that RMU possesses stable storage, after the failure its state is easily reconstructed and will not influence the processing correctness. In turn, CIMs and SIMs do not have information vital to the consistency of services that needs to survive failures. They obtain all the data from RMUs or in the course of requests' processing. Thus, the failure either of SIM or of CIM does not influence the consistency, as they are always able to recover by resubmitting requests. The system state reached during the failure-free processing is consistent. In case of failure of clients or services, and after applying the proposed rollback-recovery protocol, their recovered state corresponds to the state before the failure. Thus, the proposed rollback-recovery protocol leads to the consistent recovery of processing in the service-oriented systems, despite failures of system components.
Full versions of the theorems and proofs can be found in [6]. □

7 Conclusions

This paper has dealt with a problem of increasing the fault-tolerance of SOA systems. The rollback-recovery protocol, ensuring the recovery of a coherent state of SOA distributed processing in the case of failure of one or more system components (i.e., web services or their clients), has been proposed. Although our implementation of the protocol is based on the known technique of operation logging, it is nevertheless unique in exploiting properties of SOA while applying these solutions. Our future work encompasses the integration of the proposed rollback-recovery protocol with the consistency protocols, in order to relax the consistency model provided during the recovery. Simultaneously, the work to increase the efficiency of the proposed protocol is underway, and the appropriate simulation experiments to quantitatively evaluate the overhead of the presented rollback-recovery protocol are being carried out.

References

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[2] Jerzy Brzeziński, Dariusz Dwornikowski, and Jacek Kobusiński. FADE: RESTful service for failure detection in SOA environment. In Victor Malyshkin, editor, *Parallel Computing Technologies*, volume 6873 of *Lecture Notes in Computer Science*, pages 238–243, Kazan, Russia, September 2011. Springer Berlin.

[3] J.-Y. Chen, Y.-J. Wang, and Y. Xiao. SOA-based service recovery framework. In *Proceedings of the 9th International Conference on Web-Age Information Management*, pages 629–635, July 2008.

[4] N. Elmootazbellah, Elnozahy, A. Lorenzo, Yi-Min Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[5] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[6] Mateusz Hołenko and Piotr Zierhoffer. Protokoły odtwarzania stanu w systemach zorientowanych na usługi. Master's thesis, Poznań University of Technology, Poznań, September 2011.

[7] Jim Lau, Lau C. Lung, Joni da S. Fraga, and Giuliana Santos Veronese. Designing fault tolerant web services using BPEL. In *Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (ICIS 2008)*, pages 618–623, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Alexandros Marinos, Amir R. Razavi, Sotiris Moschoyiannis, and Paul J. Krause. RETRO: A consistent and recoverable RESTful transaction model. In *ICWS*, pages 181–188, 2009.

[9] OASIS. Web services reliable messaging (WS-ReliableMessaging) version 1.1, January 2008.

[10] Leonard Richardson and Sam Ruby. *RESTful Web Services.* O'Reilly Media, 2007.

[11] Erl Thomas. *SOA Principles of Service Design.* Prentice Hall PTR, 2007.

Received December, 2012