Towards Relaxed Rollback-Recovery Consistency in SOA

Jerzy Brzeziński, Mateusz Holenko, Anna Kobusińska, Dariusz Wawrzyniak, Piotr Zierhoffer

Towards relaxed rollback-recovery consistency in SOA*

Jerzy Brzeziński, Mateusz Hołenko, Anna Kobusińska, Dariusz Wawrzyniak, and Piotr Zierhoffer

Institute of Computing Science
Poznań University of Technology, Poland
{jbrzezinski,mholenko,akobusinska,dwawrzyniak,pzierhoffer}@cs.put.poznan.pl

Abstract. Nowadays, one of the major paradigms of distributed processing is SOA. To improve the reliability of SOA-based systems, a ReServE service that ensures recovery of consistent processing state, has been proposed. ReServE introduces a high overhead during failure-free computing. Thus, in this paper we propose relaxed recovery consistency models that allow optimization of rollback-recovery in SOA. We propose their formal definitions, and discuss the conditions under which these models are provided by ReServE.
Keywords: SOA, web services, fault tolerance, message logging, rollback-recovery, consistency

1 Introduction

In the recent years, the rapid growth of development and deployment of service-oriented systems (SOA) has been observed [8]. Although SOA-based systems have many advantages, they are also highly error-prone. Failures of SOA components, lead to limitations in the availability of services, affecting the reliability of the whole system. Such a situation is highly undesirable from the viewpoint of SOA clients, who expect that provided services are reliable and available. To improve reliability of SOA-based systems and applications, different approaches may be applied. Among them are: replication, transaction-based forward recovery (which requires the user to explicitly declare compensation actions), and the rollback-recovery checkpoint-based approach [5].

In many existing SOA systems, in case of service failure, the compensation procedure is often applied to withdraw the effects of the performed request [2,9]. However, there are situations, when compensation procedure is either impossible, or it can be prohibitively expensive. In such situations, the rollback-recovery approach [6], known from the general distributed systems can be applied. Unfortunately, the rollback-recovery techniques for general distributed systems do not into account specific properties of SOA systems, among which are: the autonomy of nodes, loose-coupling, heterogeneous nature of the environment, the

* This work was supported by the Polish National Science Center under Grant No. DEC-2011/03/D/ST6/01331

dynamic nature and the longevity of interactions, and the inherent constant interaction with the outside world. As a consequence, web services should not be forced to take a checkpoint or to roll back in case of the fault-free execution. They can also refuse to inform other services on checkpoints they have taken. Therefore, there is a need for rollback-recovery mechanisms specially tailored for SOA architectures.

Responding to this need, we proposed ReServE (Reliable Service Environment), which aims in increasing the SOA fault-tolerance [3,4]. ReServE while preserving services autonomy, ensures at the same time 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 recovered. ReServE focuses on seeking automated mechanisms that do not require the user intervention in the case of failures, and are *other* than transactions or replication. The proposed service can be used in any SOA environment, though it is particularly well-suited for the processing which does not have the transactional character, and for the applications that do not use the business process engines with internal fault-tolerance mechanisms (e.g. BPEL). It also respects the independence of the service providers, allowing them to implement their own fault-tolerant policies.

ReServE guarantees that the recovered execution is perceived by all participants of the processing in a consistent manner. Since, according to our best knowledge, the notion of a consistent recovery state has not been clearly defined and formalized in the context of the SOA, during recovery we followed the intuitive approach, by which the recovered state is said to be consistent, if it reflects the observable behavior of the system before the failure. In this paper, we clearly define and formalize the notion of a strict SOA-based recovery consistency model, implemented in ReServE until now. Because providing such a strict recovery consistency introduces a large overhead during the failure-free computing, we discuss under which conditions a strict recovery consistency can be relaxed. Consequently, we propose formal definitions of relaxed recovery consistency models that allow the recovered service state to differ from the one before the time of failure. We also determine which interactions (and in which order) have to be recovered by ReServE service, to ensure the continuation of the processing consistent accordingly to the proposed relaxed recovery consistency models.

The rest of the paper is structured as follows: section 2 presents system model and basic definitions. Section 3 describes the general idea of ReServE, which summarizes already presented service, and is included in order to make a paper self-contained. The main contribution of this paper is contained in Sections 4 and 5, where the formal definition of strict consistency model is presented, and relaxed recovery consistency models are proposed. Next, in Section 6 it is analyzed how the proposed recovery consistency models are realized within ReServE service. Finally, Section 7 concludes the paper.

2 System model and basic assumptions

Throughout this paper, SOA system model is considered. We focus on RESTful web services [10], exposed as sets of resources, and identified by a uniform resource identifier (URI) mechanism. Resources can be characterized as a set of data items, which may be simple variables, files, objects of object-oriented programming language, etc. A client may interact with such services employing the HTTP protocol operations, with their customary interpretation. Services are published by service providers $S_k \in \mathcal{S}$ and accessed by service consumers (clients) $C_i \in \mathcal{C}$. The basic interaction between a client and a service consists of service invocation (an event at the client side), and its execution (an event at the server side). The code to be executed, i.e. the implementation of service functionality is termed a *method*. Invocations and executions correspond to communication events at the protocol level. Invocation starts with sending a request message from a client to a server, and matching receipt at the server side. Execution finishes with a reply message sent from the server to the client. The receipt of the reply completes the invocation. The sequence of interactions between clients and web services will be called a business process. Both clients and services are piece-wise deterministic. Services can concurrently process only such 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.

According to the REST rules, communication in the considered system is stateless, which means that each request contains all the information necessary to understand the request, independently of any requests that may have preceded it. The considered communication channels are reliable (the reliability is ensured by retransmission of messages and appropriate filtering of duplicates), but they do not guarantee 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 [7]. Furthermore, we assume that each service provider may use different mechanisms to provide fault tolerance. By a recovery point we will denote an abstraction describing a consistent state of the service, which can be correctly recovered after a failure, but we do not make any assumptions on how and when such recovery points are made (to make a recovery point logs, checkpoints, replicas and other mechanisms may be used). Each service takes recovery points independently. Similarly, the client may also provide its own fault tolerance techniques to save its state.

3 ReServE — the general idea

In this section, the design choices and concepts behind ReServE service are presented. The detailed description of ReServE has already been presented in [3,4], and is included here in order to make a paper self-contained. Due to the

fact that interactions between clients and services result in possible resource state changes, they entail the client-service inter dependencies. Because in SOA the autonomy of services is assumed, the failure of one process should not influence the processing of other processes, and should not force them to rollback when they have not failed. Since service providers do not provide information on the internal implementation of services, it is not known which events introduce inter-process dependencies. Therefore, the recovery of a failed service should be isolated to avoid the cascading rollbacks of other processes.

The architecture of ReServE is shown in Fig. 1. It has a modular construction, and includes Recovery Management Units (*RMU*), Client Intermediary Modules (*CIM*) and Service Intermediary Modules (*SIM*). The main task of *RMU*s is increasing the reliability of performed business processes. *RMU*s store all requests and responses exchanged by business process participants in their Stable Storage able to survive all failures. As a result, the *RMU* modules posses a complete history of communication, which is used during rollback and recovery of business processes. The remaining *CIM* and *SIM* units serve as proxies for clients and servers. They make ReServE transparent to participants of the communication, and allow to fully control the flow of messages in the system by intercepting messages issued by clients and servers. Additionally, *SIM*s monitor the services status and react in the case of its failure by initiating and managing the service rollback-recovery procedure.

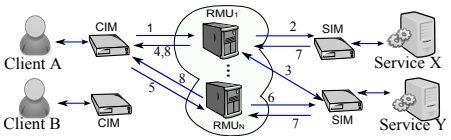

Fig. 1. ReServE architecture

Each service is registered in one *RMU* (default or master *RMU*), but the single *RMU* can be used by many services. In turn, the client can be registered simultaneously in many *RMU*s, but always one of them stores information on other *RMU*'s used by the client. The request issued by a client to a service is intercepted by client's *CIM*, and forwarded to its master *RMU* (1). If the required service is registered in *RMU*, the request is saved in *RMU*'s Stable Storage, and forwarded to the service through its *SIM* (2). Otherwise, client's master *RMU* obtains the URI of requested service *RMU* from its *SIM* (3), and sends back this information to *CIM* (4), which reissues the request to a proper *RMU* (5, 6). The service processes request and sends the response back to *RMU* (7). The response is saved in the Stable Storage and forwarded to the client through *CIM* (8). If the *RMU* module obtains the client's request, to which the response has already been saved, then saved response is sent to the client, and there is no necessity to send the request once again to the service, which provides exactly-once execution of a client's request.

4 Strict recovery consistency model

In ReServE, the consistent recovery assumes the recovery of all events that have occurred before the failure in the same order as during the original execution. In effect, the recovered service reaches the state from before the failure. This approach lacks proper theoretical foundations — according to our best knowledge, neither the notion of a consistent recovery state has been clearly defined and formalized in the context of SOA, nor the requirements of the consistency have been specified. Finding the consistent state of SOA computation is important for analyzing, testing or verifying properties of these computations. Thus, the lack of formally specified and recognized consistency requirements for SOA-compliant processing gravely prohibits the construction of provably correct rollback-recovery protocols. Therefore, this paper aims at giving the necessary formal basis for any further in-depth research in this field.

In this section the ***strict recovery consistency model*** (***AllRequests***) that corresponds to ReServE pessimistic approach is proposed. The failure occurrence in this case is fully masked, and the recovery is transparent from the viewpoint of clients and services. In the formal definition of this model, the notation presented below is used.

The set of all methods provided by a service is denoted by $\mathcal{M}$. They either *modify* (possibly also read) or *only read* the service state (they belong to sets $\mathcal{M}_{\mathrm{M}}$ and $\mathcal{M}_{\mathrm{R}}$ respectively). When a client C_i invokes a service S_j by sending the x-th request req that refers to a resource res an event denoted by $req^x_{C_i}(S_j, op, res)$ is produced. The parameter op of this event denotes the type of method ($op \in \mathcal{M}$) to be executed by the service in the result of obtaining the request. In turn, $recv\_rep^x_{C_i}(S_j, res)$ represents the event produced when a client C_i obtains from a service S_j the reply rep to its x-th request req; result of execution of req is return in res. The corresponding events at a service S_j, are: $recv\_req^x_{S_j}(C_i, op, res)$ and $rep^x_{S_j}(C_i, op, res)$. The former denotes the event produced when S_j receives the appropriate request from C_i. The latter represents the event produced when the service S_j has finished the execution of the request , and sends a reply to C_i. For the sake of the simplicity, if some element in the above notation is unimportant or obvious in the context, it can be omitted. The local history of a service S_j is denoted by $H_{S_j} = E^0_j E^1_j E^2_j \ldots E^n_j$. Events that occur at service S_j are ordered by relation $\overset{S_j}{\rightarrowtail}$, called *service execution order*. In turn, the relation of events that occurs during service recovery is represented by $\overset{S_j}{\twoheadrightarrow}$, and is called *service recovery execution order*.

Each time when the failure of service S_j occurs, a crash event denoted by f^* is produced. In turn, in a moment of recovery a restart event $\hat{f}$ occurs. Thus, service state at the moment of event $\hat{f}$ occurrence is equivalent (in the result of the performed rollback) to the state saved in the latest recovery point RP_{S_j}. We denote the local history of a service S_j comprising events that occurred after the service recovery point RP_{S_j} was taken, but before the crash event f^* by $H^{\prec f^*}_{S_j}$. In turn, the local history of service S_j comprising events that occurred after

the restart event is denoted by $H_{S_j}^{\succ \hat{f}}$. Consequently, $rep_{S_j}(C_i, op, res) \in H_{S_j}^{\prec f^*}$ denotes the event of sending reply lost due to the failure by a service S_j to a client C_i, and $recv\_req_{S_j}(C_i, op, res) \in H_{S_j}^{\succ \hat{t}}$ indicates that the event of receiving request by service S_j from client C_i was recovered after the restart event $\hat{f}$.

Informally, the recovered service state is said to be consistent according to strict recovery consistency model, if after recovery from a failure, the service state reflects the execution of all requests obtained from clients and other services, and performed by this service before its failure. Moreover, the order of recovered requests is the same as it was before the failure. Below, the formal definition of AllRequests recovery consistency model is presented:

Definition 1. *Let $o1, o2 \in \mathcal{M}$ be methods provided by a service S_j. The recovered service state is consistent according to **strict recovery consistency model (AllRequests)**, iff for all events $recv\_req_{S_j}(C_i, o1)$, $recv\_req_{S_j}(C_k, o2)$ that represent requests obtained by service S_j, and $rep_{S_j}(C_i, o1)$, $rep_{S_j}(C_k, o2)$ that are replies issued by S_j after performing methods $o1$ and $o2$, the following condition holds:*

$$rep_{S_j}(C_i, o1) \in H_{S_j}^{\prec f^*} \Rightarrow recv\_req_{S_j}(C_i, o1) \in H_{S_j}^{\succ \hat{f}} \wedge$$
$$\forall rep_{S_j}(C_i, o1), rep_{S_j}(C_k, o2) \in H_{S_j}^{\prec f^*} :: rep_{S_j}(C_i, o1) \overset{S_j}{\rightarrowtail} rep_{S_j}(C_k, o2) \Rightarrow$$
$$recv\_req_{S_j}(C_i, o1)) \overset{S_j}{\twoheadrightarrow} recv\_req_{S_j}(C_k, o2)$$

The above definition says that if a method $o1$ was performed before the service failure (i.e. the event of sending a reply after performing$o1$ belongs to history $H_{S_j}^{\prec f^*}$ of events performed by S_j before the failure), then the method $o1$ is recovered. This implies that the event of receiving request that invokes $o1$ is applied again after the service restart, and belongs to history $H_{S_j}^{\succ \hat{f}}$ of events performed by service S_j after its rollback. Moreover, if replies to $o1$ and $o2$ were issued before the failure in a specified service execution order, their execution order is the same during the recovery. A formal specification allows unambiguous determination of the set of requests that can not be missed during the recovery, because they are necessary to meet the recovery consistency model.

5 Relaxed recovery consistency models

Service providers supply clients with a set of methods that allow them to benefit from the functionality offered by services. Depending on the characteristics of a service and the nature of its methods, the execution of these methods differently affects the service state. Some methods are past-operations-aware, i.e. they take into account the history of service processing in order to modify the service state, whereas the execution of other methods invalidates the previous service history, or part of it. Therefore, although some methods modify the service state, they

are *irrelevant* to the overall service computation due to the method specificity. To illustrate this, let us consider a counter service that provides the following methods: $inc(x)$ that increases a value of the counter resource by x, $dec(x)$ — decreasing a value of the counter by x, and $set(x)$, which sets the counter value for x. Further, let us assume that the following sequence of methods was performed: $inc(5)$, $dec(3)$, $set(7)$, $dec(1)$. After the failure occurrence only methods $set(7)$, and $dec(1)$ have to be recovered, because the result of execution of methods $inc(5)$, and $dec(3)$ was overridden by the execution of a method $set(7)$.

Re-execution of irrelevant methods can be omitted during the rollback-recovery, without changing the meaning of processing and its result. Consequently, such methods also need not to be logged. This implies that some services do not require a strict recovery consistency model to recover processing perceived as consistent. Below, we propose relaxed recovery consistency models that allow optimization of the rollback-recovery. In order to alleviate requirements regarding the consistent processing state, we assume that service provider delivers basic information on the character of methods it executes during service processing.

Every modification recovery consistency model Lessons learned from the message-passing systems, in which read messages are neglected during the process rollback-recovery, enabled us to divide methods into lookup and modifying. Methods from the first group do not change the state of a service, so they can be considered irrelevant from the service point of view, and as such they can be omitted during the recovery of a service state. In turn, all modifying methods performed before the failure have to be recovered in the case of a service failure. Moreover, the order of their execution before the failure have to be maintained after the recovery. A recovery consistency model that ensures this assumption is called ***every modification recovery consistency model*** (***EveryMod***).

Definition 2. *Let $o1, o2 \in \mathcal{M}_M$ be modifying methods provided by a service S_j. The recovered service state is consistent according to **every modification recovery consistency model (EveryMod)**, iff for all events $recv\_req_{S_j}(C_i, o1)$, $recv\_req_{S_j}(C_k, o2)$ that represent requests obtained by service S_j, and $rep_{S_j}(C_i, o1)$, $rep_{S_j}(C_k, o2)$ that are replies issued by S_j after performing $o1$ and $o2$, the following condition holds:*

$$rep_{S_j}(C_i, o1) \in H_{S_j}^{\prec f^*} \Rightarrow recv\_req_{S_j}(C_i, o1) \in \hat{H_{S_j}^{\succ f}} \wedge$$
$$\forall rep_{S_j}(C_i, o1), rep_{S_j}(C_k, o2) \in H_{S_j}^{\prec f^*} :: rep_{S_j}(C_i, o1) \overset{S_j}{\rightarrowtail} rep_{S_j}(C_k, o2) \Rightarrow$$
$$recv\_req_{S_j}(C_i, o1)) \overset{S_j}{\twoheadrightarrow} recv\_req_{S_j}(C_k, o2)$$

EveryMod recovery consistency model loosens AllRequests model by taking into account only operations that modify service state ($o1, o2 \in \mathcal{M}_M$) instead of all operations performed before the failure. EveryMod can be applied to all e-commerce services. Let us consider an on-line store. Purchasing or returning products bought in this store changes the amount of available products and the

store's budget. After a service failure, all performed purchases and returns have to be recovered. On the other hand, when a client just checks if the item is offered by on-line store or how much it costs, then the request corresponding to above method can be omitted during recovering a service state.

Important modification recovery consistency model Let us consider that among modifying methods provided by a service, there is a set of methods that are significant for providing a service functionality. The execution of such methods does not take into account the history of other, previously performed methods. Therefore, during the rollback-recovery only significant methods have to be recovered.

A recovery consistency model that differentiates service modifying methods, and distinguishes a set of methods significant for supplying service functionality is called ***important modifications recovery consistent model*** (***ImpMod)***. The execution of significant methods does not take into account the history of previously performed methods, which are not significant. Informally ImpMod recovery consistency model implicates that all requests of significant methods have to be recovered. The execution order of recovered requests corresponds to their execution order before the failure occurrence. When significant methods have not been executed before the failure, then all modifying requests have to be recovered. Finally, requests modifying service state invoked after the execution of the last request of a significant method also have to be recovered.

Definition 3. Let $o \in \mathcal{M}_{\mathrm{M}}$ *be modifying methods provided by a service* S_j*, and* $o' \in \mathcal{M}_S$ *be significant methods provided by* S_j *, where* M_S *denotes the set of significant methods* $M_S \subset M_M$*. Further let* $o1, o2$ *be methods of the same type* ($o1, o2 \in \mathcal{M}_S$ *or* $o1, o2 \in \mathcal{M}_M$)*. The recovered service state is consistent according to* ***ImpMod recovery consistency model****, iff for all events* $recv\_req_{S_j}(C_i, o)$*, that represent requests obtained by service* S_j*, and* $rep_{S_j}(C_i, o)$ *that are replies issued by* S_j*, the following condition holds:*

$$\begin{gathered}
\left(\forall rep_{S_j}(C_i, o') :: rep_{S_j}(C_i, o') \in H_{S_j}^{\prec f^*}\right) \Rightarrow recv_req_{S_j}(o') \in \hat{H_{S_j}^{\succ f}} \wedge \\
\forall rep_{S_j}(C_i, o) \in H_{S_j}^{\prec f^*} :: \\
\left[\left(\nexists rep_{S_j}(C_i, o') \in H_{S_j}^{\prec f^*} :: rep_{S_j}(C_i, o) \overset{S_j}{\rightarrowtail} rep_{S_j}(C_i, o')\right) \Rightarrow recv_req_{C_i}(o) \in \hat{H_{S_j}^{\succ f}}\right] \wedge \\
\left(\forall recv_req_{S_j}(C_i, o1), recv_req_{S_j}(C_k, o2) \in H_{S_j}^{\prec f^*} :: rep_{S_j}(C_i, o1) \overset{S_j}{\rightarrowtail} rep_{S_j}(C_k, o2)\right) \\
\Rightarrow \left(recv_req_{S_j}(C_i, o1) \overset{S_j}{\twoheadrightarrow} recv_req_{S_j}(C_k, o2)\right)
\end{gathered}$$

Above definition states that every significant method performed before the failure is recovered, because when the reply issued after the execution of significant method o' belongs to history $H_{S_j}^{\prec f^*}$ of events performed by S_j before the failure, then the request of method $o1$ is re-invoked by S_j after its restart,

and belongs to $\hat{H_{S_j}^{\succ f}}$ (first condition). Further, it is said that all modifying methods o executed before the failure (for which $rep_{S_j}(C_i, o) \in H_{S_j}^{\prec f^*}$) that were not followed by any significant method o'. In other words, if there exists no significant method o' invoked after the invocation of modifying methods o: $\left(\nexists rep_{S_j}(C_i, o') \in H_{S_j}^{\prec f^*} :: rep_{S_j}(C_i, o) \overset{S_j}{\rightarrowtail} rep_{S_j}(C_i, o')\right)$, then the invocation of modifying methods is recovered and belongs to the history $\hat{H_{S_j}^{\succ f}}$ (second condition). Finally, the execution order during recovery procedure corresponds to the execution order before the failure.

To illustrate the application of ImpMod recovery consistency model let us assume that a service provides clients a virtual shopping basket, and supplies methods to operate on it (*add* and *remove*), as well as to finalize electronic shopping (*buy*). When a client adds or removes products from the basket, the amount of available products changes, what is reflected in the state of the on-line store. After the failure occurrence, when the on-line store restarts its work, the shopping basket of a client should comprise all products that have been added to it before the failure (the history of methods performed by a client consists of a sequence of *add* and *remove*). However, when the client finalizes its shopping the shopping basket is emptied. After recovery the history of performed actions contains only the information on finalizing shopping (there is just a buy *method*).

Latest modification recovery consistency model Among modifying methods there can be distinguished those that override the service state, without taking into account the prior history of states. In such case, only the latest executed method is essential for the proper recovery of the service state, and as such it should be persistent. A recovery consistency model that ensures this assumption is called ***latest modification recovery consistency model*** (***LatestMod***).

Definition 4. *Let $o1, o2 \in \mathcal{M}_L$* be *modifying methods that belong to the set $\mathcal{M}_L$ of methods that override a service state, where $M_L \subset M_M$. The recovered service state is consistent according to **LatestMod recovery consistency model**, iff for all events $recv\_req_{S_j}(C_i, o1)$, $recv\_req_{S_j}(C_i, o2)$ that represent requests obtained by service S_j, and $rep_{S_j}(C_i, o1)$, $rep_{S_j}(C_i, o2)$, that are replies issued by S_j, the following condition holds:*

$$\left(\forall rep_{S_j}(C_i, o1),\, rep_{S_j}(C_i, o2) \in H_{S_j}^{\prec f^*} :: rep_{S_j}(C_i, o1) \overset{S_j}{\rightarrowtail} rep_{S_j}(C_i, o2)\right) \Rightarrow \left(recv\_req_{S_j}(C_i, o2) \in \hat{H_{S_j}^{\succ f}}\right)$$

The key difference between ImpMod and LatestMod recovery consistency models consists in recovering only the single, latest request performed by the service before the failure in the case of the LatestMod mode. In contrast for the LatestMod recovery consistency model, in ImpMod a set of requests is recovered.

Continuing the example of the on-line store, let us assume that a client of the on-line store manages its client's account profile. A client can change his/her personal details. Every modification of the client account is binding, so only the latest modification one is recovered.

No modifications recovery consistency model In case of some services, the modifying methods can be unheralded from the viewpoint of such services. This is a case, of all services that mediate in the execution of methods requested by a client, and act as proxy services. Such services refer requests from clients to appropriate services providing functionality required by clients. A recovery consistency model that refers to intermediary services, that only mediate in the processing between clients and other services, is called ***no modification recovery consistency model*** (***NoMod***).

Definition 5. *Let* $o1, o2 \in \mathcal{M}_M$ *be modifying methods. The recovered service state is consistent according to* ***NoMod recovery consistency model****, iff for all events* $recv\_req_{S_j}(C_i, o1)$, $recv\_req_{S_j}(C_i, o2)$ *that represent requests obtained by service* S_j*, and* $rep_{S_j}(C_i, o1)$, $rep_{S_j}(C_i, o2)$, *that are replies issued by* S_j*, the following condition holds:*

$$\left(\forall rep_{S_j}(C_i, o1),\, rep_{S_j}(C_k, o2) \in H_{S_j}^{\prec f^*}\right) \Rightarrow \hat{H_{S_j}^{\succ t}} = \emptyset$$

6 Discussion on the consistent recovery problem

In this section we discuss the realization of the proposed recovery consistency models in the context of ReServE. In order to recover a service state that is consistent according to a required recovery consistency model, a set of requests that should be re-executed after the service failure has to be designated. Also, the order in which the chosen requests are performed during the service recovery has to be determined. Therefore ReServE service makes some necessary assumptions, and introduces internal mechanisms, to solve this problem. We discuss them briefly below.

In order to provide the correct recovery, the requests should be re-executed in the appropriate order. For this purpose, each reply sent by the service has a unique identifier, called *ResponseId*, which is assigned by a service. Relying on the provided formal specifications of the models presented in section 5, we determine which messages have to be kept by the RMU and resent to a service, during its recovery. Since AllRequests model is the most general recovery consistency model, we only describe the way it differs from other models. Moreover, actions performed by other modules (specifically SIM) are the same for all consistency models and are described in [3,4].

AllRequests recovery consistency model requires all requests to be saved. Only when a service informs about a new recovery point, RMU is allowed to remove older messages. Having received a request to start recovery process beginning with a certain message, denoted *lowestReqId*, RMU resends a set of

messages determined by the following predicate:

$$toRecover = \{req_{S_j} : (rep_{req_{S_j}} \in SavedReplies \wedge rep_{req_{S_j}}.ResponseId \geq lowestReqId) \vee (rep_{req_{S_j}} \notin SavedReplies)\}$$

The *toRecover* predicate chooses all messages directed to the given service, for which a reply has been saved with identifier greater or equal to the one requested by the service. Also all requests without an answer kept by RMU are chosen to be resent. EveryMod recovery consistency model differs from the strict one in two aspects. Firstly, a receipt of reply allows RMU to forget the content of the corresponding request. However, for the sake of a client recovery the reply must be still kept by ReServE. The request, on the other hand, will never be used again, so it's content can be safely discarded. To preserve a possibility to recover client states, the metadata of the requests has to be retained. After a failure, a set of messages to be re-executed is described by:

$$toRecover = \{req_{S_j} : req_{S_j}\}.TheContent \neq \emptyset \wedge ((rep_{req_{S_j}} \in SavedReplies \wedge rep_{req_{S_j}}.ReplyId \geq lowestReqId) \vee (rep_{req_{S_j}} \notin SavedReplies))$$

RMU chooses requests directed to the given service in a similar fashion to AllRequests algorithm, but now it omits the requests without any content, as they were deemed irrelevant to the recovery process. ImpMod recovery consistency model is a specific version of EveryMod model. Both models consider only modifying requests, but ImpMod allows RMU to reduce the amount of repeated messages even more. Upon receiving of a reply, RMU verifies if the corresponding request was modifying. If not, it's content is discarded, as in EveryMod model. If the request was modifying, it's importance, declared by the service, is verified. Receipt of a reply to an important requests causes RMU to discard content of previous unimportant requests directed to the same resource. A set of messages to recover after a failure is calculated in the same way as in the EveryMod model. Since the content of unimportant requests was removed, they won't become a part of the recovery process. For even simpler services, supporting LatestMod recovery consistency model, there is no concept of important modifications. Instead, after receiving a reply to a modifying request, RMU clears the content of a previous request directed to the same resource. A response to a non-modifying request, as in previous models, triggers clearing of this request's content. This way there is at most one request saved for each resource.

7 Conclusions

Although some attempts to increase the fault-tolerance of SOA systems have been undertaken, the proposed solutions, based on rollback-recovery mechanism, require costly global recovery coordination, offering very strict consistency of the recovered processing state. It is clear, based on the past experience, that many SOA applications could benefit from less restrictive consistency models, allowing

the recovery of the processing state in a more efficient way. But, according to best authors knowledge, neither the notion of a consistent recovery state has been clearly defined and formalized in the context of SOA, nor the requirements of the consistency have been specified. Therefore, this paper has dealt with a problem of providing consistency models for rollback-recovery of SOA systems. In the paper, the formal definitions of recovery consistency models were proposed, and their features were discussed. A formal specifications allowed the unambiguous determination of the set of requests that can not be missed during the recovery. The proposed recovery consistency models were applied in the context of ReServE service. Our future work encompasses carrying out the appropriate simulation experiments to quantitatively evaluate the overhead of the presented relaxed rollback-recovery protocols.

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. Luis Felipe Cabrera, George Copeland, Bill Cox, Tom Freund, Johannes Klein, Tony Storey, and Satish Thatte. Web services transactions specifications, 2005.
3. Arkadiusz Danilecki, Mateusz Hołenko, Anna Kobusińska, Michał Szychowiak, and Piotr Zierhoffer. ReServE service: An approach to increase reliability in service oriented systems. In Victor Malyshkin, editor, *Proc. of the 11th Int. Conf. on Parallel Computing Technologies*, volume 6873, pages 244–256, Kazan, Russia, September 2011. Springer Berlin.
4. Arkadiusz Danilecki, Mateusz Hołenko, Anna Kobusińska, Michał Szychowiak, and Piotr Zierhoffer. Applying message logging to support fault-tolerance of SOA systems. *Foundations of Computing and Decision Science*, 38(3):145–158, 2013.
5. Vijay Dialani, Simon Miles, Luc Moreau, David De Roure, and Michael Luck. Transparent fault tolerance for web services based architectures. In *Proc. of the 8th International Euro-Par Conference (Euro-Par 2002)*, number 2400 in Lecture Notes in Computer Science, pages 889–898, Paderborn, Germany, August 2002. Springer-Verlag.
6. 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.
7. Michał Kalewski, Anna Kobusińska, and Jacek Kobusiński. FAST failure detection service for large scale distributed systems. In *Proc. of the 17th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP 2009)*, pages 229–236, Weimar, Germany, February 2009. IEEE Computer Society.
8. Ken Laskey, Jeff A. Estefan, Francis G. McCabe, and Danny Thornton. *Reference Architecture Foundation for Service Oriented Architecture Version 1.0 Committee Draft 02*. OASIS, 2009.
9. 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.
10. Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, 2007.