Reliability Service for Service Oriented Architecture*

Arkadiusz Danilecki, Mateusz Hołenko, Anna Kobusińska, Michał Szychowiak, and Piotr Zierhoffer

Institute of Computing Science
Poznań University of Technology, Poland
{Arkadiusz.Danilecki,Anna.Kobusinska,Michal.Szychowiak}@cs.put.poznan.pl,
{mateusz.holenko,piotr.zierhoffer}@gmail.com

Abstract. Nowadays, a major paradigm of large scale distributed processing is service-oriented computing (or SOA, Service Oriented Architecture). To improve availability and reliability of SOA based systems and applications, a Reliability Service (providing external support of web services recovery) is proposed.
Keywords: fault tolerance, rollback-recovery, RESTful web services

1 Introduction

A service-oriented architecture allows to integrate new or legacy applications and to expose them via web interface as web services [5]. Such services may cooperate with services from other providers. Therefore, SOA systems are susceptible to faults, which are unavoidable in any large scale, distributed system. There are many solutions of fault tolerance problem, however, most of them are not well profiled for SOA systems due to their specific characteristics, among which are: the autonomy of the service providers; dynamic nature of the interaction; longevity of interaction; and the inherent constant interaction with the outside world. Thus, there is a need for fault tolerance mechanisms specially tailored for SOA.

The Reliability Service presented in this paper is specifically confined to find the solution of fault tolerance problem using mechanisms *other* than transactions or replication. This requirement has dictated several design decisions. For example, we do not intend to implement compensation mechanisms. In order to support services during their recovery, The proposed service logs messages and replays the requests and/or responses in the case of failures. The elements of Reliability Service infrastructure

* The research presented in this paper was partially supported by the European Union in the scope of the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

are built according to constraints of the Resource Oriented Architecture [4], which follow the constraints of REST paradigm [1]. Currently the Reliability Service supports only RESTful web services, however in the future we intend to support also services with SOAP interfaces (also known as Big Web Services).

The rest of the paper is structured as follows: the system model and basic definitions are presented in Section 2. Section 3 describes the proposed Reliability Service: its architecture and functionality. Finally, Section 4 presents areas of possible future work and concludes the paper.

2 System Model and Basic Definitions

A fundamental SOA element is a *service*: autonomous, platform - independent computational element with well-defined interface that implements some business functionality and can be described, published, discovered and accessed over the Internet using standard protocols. Services, created and maintained by service providers (SP), are used by service consumers (SC). A client requesting access to the service may not know in advance the identity of the SP, which will handle the request. The service may be *compound*, i.e. built of other services (which can also be compound).

Business process is a sequence of interactions between many web services that are performed to achieve business objectives. The definition of a business process specifies the behavior of its participants (client applications and services) and describes the ordering of service invocations. During the interaction, an obligation (i.e. a promise of an action in the future) may be established. The obligation can be undone only explicitly with compensation procedures. Business process definition specifies when during an interaction an obligation is established. In the proposed service we assume any information on business process definition, and in the consequence we assume that every message may transmit an obligation.

In the paper, it is assumed that both, CPs and SPs are expected to be piece-wise deterministic, i.e. they should generate the same results (in particular, the same URIs for a new resource) in the result of multiple repetition of the same requests, assuming the same initial state. Additionally, the crash-recovery model of failures is assumed [2]. System components can fail at arbitrary moments, but every failure is eventually detected, for example by the failure detection service. The failed SP becomes temporally unavailable until it is restored. The state of SP, which can be correctly reconstructed after a failure is called a *recovery point*. In order to create recovery points, logs and periodic checkpoints may be used. We

assume that SP makes the decision to take a checkpoint independently and in general, it may take no checkpoints at all. Similarly, CP may or may not save its state. It is important to emphasize that the proposed service does not dictate checkpoint policies neither to SP, not to CP.

3 Business Process Reliability Service

3.1 Reliability Service Architecture

The Reliability Service architecture is presented in Figure 1. The main module of the Service is the *Recovery Management Unit* (*RMU*) (conceptually, *many* different *RMU*s may exist, but in the current prototype there is only one). Other two modules are proxy servers: *Client Proxy Unit* (*CPU*) and *Service Proxy Unit*(*SPU*). Their role is to hide the service architecture details from clients and services, respectively. Any service at any moment may call other services; such a service becomes a client itself and as a consequence, it has also its own *CPU* apart from the *SPU*.

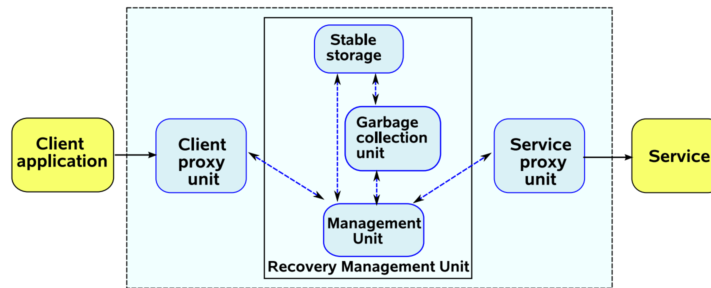

Fig. 1. Reliability Service Architecture

The *RMU* records all invocations and responses sent between clients and services. In order to ensure the proper load balancing and high availability, the *RMU* will be replicated. Three main modules of the *RMU* are: *Stable Storage* (*SS*), implemented over a database, *Management Unit (MU*) and *Garbage Collection Module* (*GCM*). A *GCM* module prevents the amount of data held by *SS* to grow indefinitely, by removing the information not used any longer. Because of the space limitations, in this paper we do not further explain this module's role.

The role of the *CPU* is to intercepts all requests issued by a client, to modify these requests accordingly to the *RMU* requirements, and to

send them to the *RMU*. We assume that the *CPU* fails together with the client; the simultaneous failures are forced if necessary.

The *SPU* is located at the service provider site. It's primary task is monitoring the service and responding to service failures. In the case of failure (detected by Service Oriented Failure Detection Service [3]), the *SPU* is responsible for initiating and managing the rollback-recovery process. The *SPU* serves a role of façade for the service: the service should be available only via the *SPU*. There is exactly one *SPU* per service. Clients may use only registered services. We assume reliable communication link between the *SPU* and the service.

3.2 The execution of a business process with the Reliability Service

When the user logs in to the client application it is asked for its identifier. The *RMU* may be then contacted to get the last saved client's state (which could be written during logout from other machine). All requests first pass through client's *CPU*, that forwards them to the *RMU*.

If the request is not guaranteed to be read-only then the *RMU* saves it in its *SS* module. The request is then forwarded to the *SPU*, which then passes it to the service. The service receives a request and executes it in accordance with its business logic. *SPU* intercepts a response sent from the service. The `Response-Id` identifier attached to the response reflects the order in which response was generated by the service. *SPU* forwards response to the *RMU*.

RMU uses `Response-Id` to force FIFO on communication link to the *SPU* (i.e the responses are postponed until all previous responses arrived to). The response is logged in the *SS* module. Additionally, the response is also stored as the last response sent to the client who initiated the request, and the value `Response-Id` is stored as the `Last-Response` received from the service. Once this is done the response is passed to the *CPU*. The *CPU* removes all custom HTTP headers added by the Reliability Service components before passing it to the client.

The messages may be retransmitted periodically; the duplicates are detected by *RMU*, *SPU* and *CPU* use set of message identifiers. Therefore, *exactly once* delivery is guaranteed in the case of failure-free execution.

In the case of client's application failure, for some clients the last response from the service may be enough for recovery (according to the HATEOAS principle of Resource Oriented Architecture). Such clients may

start by contacting with *RMU* and requesting the last response stored by *RMU*. Then they directly proceed with the execution.

If the last response is not sufficient for client's revival, such a client must cooperate with Reliability Service. At the beginning, the client should first recover using its own local checkpoints or logs. Later client sends requests to the *CPU* as usual. The requests are forwarded to the *RMU*. If the *RMU* already has the response for the request, it sends such a response to the client. The duplicate requests are ignored by *RMU*. Since the client is piece-wise deterministic, it should reconstruct its state up to the point of the last request sent before the failure. Finally the client sends a request for which *RMU* has no stored response and which was not received in the past by the *RMU*. This indicates that client's recovery is finished at this point, and *RMU* starts to forward the request to *SPU*.

Once service's failure is detected, or if the *SPU* fails itself, the service's rollback-recovery process starts. At the restart, the *SPU* asks the service for the list of available recovery points, along with the `Saved-Response` identifier. If the last recovery point contains responses, they are first sent to the *RMU*, and *SPU* waits until they are acknowledged by *RMU* before proceeding. Afterwords, the *SPU* asks the *RMU* for the `Last-Response` identifier.

The service must be rolled back to the latest recovery point for which the `Saved-Response` is less from or equal to the received `Last-Response`. Later the *SPU* asks *RMU* for a sequence of requests, informing *RMU* about value of the `Saved-Response` of the chosen recovery point. The *RMU* selects from its *SS* module all requests with no response or for which the response contains the identifier `Response-Id` greater than or equal to the `Saved-Response` value received from the *SPU*. If *RMU* has a response for a request and its `Response-Id` is less than `Last-Response`, the `Response-Id` is attached to the request, in order to inform the *SPU* on the original order of request execution.

After receiving requests from the *RMU*, the *SPU* may start the recovery. First the requests with a `Response-Id` are passed sequentially to the service, in order of their `Response-Id`. The *SPU* waits for response before sending next request. Finally the remaining requests are executed, in any order. Any error during the service or client recovery cause signal an exception to upper layers, which should then solve them using its own logic, e.g. using compensation procedures.

We require any interaction with the outside world to be modeled as a call to an external service. It's the service responsibility to ensure that such interactions are not repeated. A failure or forced rollback of one service should not force other services to rollback. In order to achieve this goal, Reliability Service stalls the calls to external services whenever there is a possibility that failure of a service could force rollback of other services.

4 Conclusions and the Future Work

This paper describes the Reliability Service providing support to web services reliability. It respects web services local recovery autonomy, and does not force any particular technique to create service recovery points. It allows to recreate lost service states in the case, when the local recovery policy is unable to achieve this (e.g. with damaged checkpoint files, or obsolete service replicas), at the cost of forcing few restrictions on service behavior. For the moment being, the prototype of the proposed service is under constant improvement: the RMU neither requires nor needs any preliminary knowledge on the service structure, business process definition, or application logic (e.g. which messages transmit obligations). Many optimizations are possible if the RMU would have an access to such informations. Moreover, transparency of the Reliability Service could be enhanced. The future work on the Reliability Service assumes further improvement of the proposed service efficiency. Moreover, a development of the mechanisms increasing the service transparency and replication mechanisms of the RMU will be carried out. Finally, minimizing recovery time and reducing failure-free overhead is needed.

References

1. Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.
2. Rachid Guerraoui and Luis Rodrigues. *Introduction to distributed algorithms.* Springer-Verlag, 2004.
3. Jarosław Kamiński, Marcin Kaźmierczak, Jacek Kobusiński, Szymon Nowacki, and Krzysztof Rosiński. Failure detection mechanism in SOA environments. Technical Report TR-ITSOA-OB1-4-PR-09-02, Institute of Computing Science, Poznań University of Technology, May 2009.
4. Leonard Richardson and Sam Ruby. *RESTful Web Services.* O'Reilly, 2007.
5. Erl Thomas. *SOA Principles of Service Design.* Prentice Hall PTR, 2007.