Replication of Recovery Log — An Approach to Enhance SOA Reliability

Anna Kobusińska(✉) and Dariusz Wawrzyniak

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

Abstract. Along with development of SOA systems, their requirements in terms of fault-tolerance increase and become more stringent. To improve reliability of SOA-based systems and applications, a ReServE service, providing an external support of web services recovery, has been designed. In this paper we propose to enhance the resilience of ReServE by replication of log with recovery information, and address problems related to deployment of this solution.

Keywords: SOA · Rollback-recovery · Message-logging · Replication

1 Introduction

Service-oriented systems (SOA) are increasingly adopted by industry in various areas of computing. Many web services, especially those found in critical or vital domains (e.g. healthcare, finance, defense, etc.), have stringent requirements in terms of availability and reliability. Since in most cases failures of such services are unacceptable, their dependability has to be ensured [1]. However, building a dependable SOA systems is a difficult task, due to their specific properties. SOA systems are highly dependent on the remote web service components of various characteristics, which are autonomous and loosely-coupled. Web services usually run on heterogeneous platforms, and are hosted by different organizations. Such services may be unavailable for an unknown reason, and for an undetermined amount of time. Moreover, their providers may refuse or be unwilling to cooperate with other providers to overcome failures of their services. They also may not be able to take part in fault-tolerance processing because of applied fault-tolerance policies. Therefore, services should not be relied upon, when the fault-tolerant mechanisms are to be provided.

As a consequence, we have proposed ReServE service, which aims in increasing SOA fault-tolerance [2]. ReServE provides an external support of web services rollback-recovery with the use of a well-known mechanism of message-logging [5,9], and ensures that in the case of failure of one or more system components (i.e. web services or their clients) a consistent state of distributed

This work was supported by the Polish National Science Center under Grant No. DEC-2011/03/D/ST6/01331 and the Grant No. 09/91/DSPB/0571.

A. Bessani and S. Bouchenak (Eds.): DAIS 2015, LNCS 9038, pp. 152–157, 2015.
DOI: 10.1007/978-3-319-19129-4_12

processing is recovered. The interactions exchanged between clients and services are saved by ReServE in the form of message log, stored in the persistent storage, which is assumed to survive all failures. Such an assumption is difficult to be guaranteed in real-life [8]. Additionally, although a persistent storage supports reliability, due to significant MTTR (mean time to recovery), its crash results in log unavailability for a substantial period of time. This way, another point vulnerable to crash is introduced, which can reduce system availability. Therefore, in this paper we propose to implement the persistent storage used in ReServE service as a replicated log containing the recovery information. Along with the replication of a message log, also the logic of the module of ReServE, called Recovery Management Unit (RMU), which is responsible for the implementation of a recovery process has to be replicated. Although a general idea of log and RMU replication is straightforward [7], and relies on storing messages necessary for recovery of system participants in replicas, its implementation raises several problems, which have to be solved. Among them are: synchronization of logs kept by different replicas, handling replicas failures, combining the replication of recovery information with the recovery processing. In this paper we discuss how the above problems may be resolved in the context of ReServE.

The paper is organized as follows. Sections 2 and 3 present system model and general idea of ReServE, respectively. Section 4 discusses the possible approach to replication of ReServE recovery log. Finally, Section 5 concludes the paper and presents the future directions of our work.

2 System Model

Throughout this paper, a distributed SOA system is considered [6]. It consists of service providers that keep resources, and deliver — in the form of provided web services — a specified functionality to clients. Web services are autonomous and loosely-coupled. They have a well-defined and standardized interface that defines how to use them. Clients invoke services by sending requests, so service invocation results in a computation, subsequent reply to the requesting client, and possible resource state changes. Service execution may also encompass the collaboration of other services (without compromising the autonomy of each individual service). It is assumed that 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. Communication in the considered system is stateless — 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 the retransmission of messages), 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 [3]. We assume that each service provider may have its own reliability policy and may use different local mechanisms that provide fault tolerance.

3 ReServE Architecture

In this Section, the design choices and concepts behind ReServE service are presented. The detailed description of ReServE has already been presented in [2,4], and is summarized here in order to make a paper self-contained. Due to the fact that interactions between clients and services result in a computation and possible resource state changes, they entail the client-service inter-dependencies. Upon a failure of one of interacting processes, such dependencies may force other processes that did not fail to rollback. Otherwise, states of processes could reflect situations impossible in any correct failure-free execution. Due to SOA assumption on autonomy of services, the failure of one process should not influence the processing of the others. Since service providers do not provide information on the internal implementation of services, it is not known which events introduce inter-process dependencies and result in state changes. Therefore, in general, the recovery of a failed service should be isolated to avoid the cascading rollbacks of other processes. Above observation had an impact on the concept of ReServE functionality. ReServE intercepts the communication between processes and logs all performed interactions (requests of service invocations and the appropriate replies) in a persistent storage of *Recovery Management Unit* (*RMU*). The intercepted messages reflect the complete history of communication, which is used to recover the consistent system state in the case of failure. However, since in SOA participants of processing may have their private mechanisms providing reliability, their state after the failure may be partially reconstructed with the use of local mechanisms. Therefore, only those messages, the processing of which was not reflected in services' (clients') recovered state, should be processed again. The task of *RMU* is to find such messages, and reissue them to the service in the same order as before the failure. After re-execution of recovered requests *RMU* intercepts replies from the service, because they have already been sent to clients and other services during the failure-free execution. *RMU* module ensures also the idem potency of obtained requests. If it obtains the client's request, to which the response has already been saved in its persistent storage, then such a saved response is sent to the client immediately, without the need of sending the request to the service once again. Thus, the same message (i.e., the message with the same identification number) may be send by a client multiple times, with no danger of multiple service invocations. Another two modules of ReServE service are *Service Intermediary Modules* (*SIM*) and *Client Intermediary Module* (*CIM*). *CIM* and *SIM* serve as proxies for clients and servers and hide the details of rollback-recovery. 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. Additionally, *SIM* monitors the services' status and react in the case of its eventual failure by initiating and managing the service rollback-recovery.

4 Replication of Recovery Information

In this Section we propose to replicate the *RMU* module of ReServE service, instead of using the persistent storage. From the perspective of clients and services, introducing *RMU* replication is transparent. Each service has a dedicated replica of *RMU* module, in which it is registered, called a *Leader.* Analogically, each client has a default *RMU* replica, called *Interceptor*. We assume that each request issued by a client is first replicated, before it is sent to the service. Analogically, the service reply is replicated, before it is sent back to the client.

The crucial issue arising from replication is consistency. Since the replication in the context of this paper concerns log, i.e. a set of requests and replies, and adding elements to a set is commutative, i.e. it does not pose a risk of conflicts, thus consistency maintenance boils down to preserving replica completeness. The completeness is important for message safety in the sense of the ability to survive *RMU* replica crash. We assume that a message is safe if it can be obtained by a given number N of replicas, despite the crash of some log servers. A number N ranges from 1 to $\mid RMU \mid$. When $N = \mid RMU \mid$, all correct replicas hold the message, and the highest level of message safety is achieved. At the same time the system availability (response time), is decreased because before the message is sent to its recipient, first all N replicas have to acknowledge the fact that they obtained the message. In turn, in the worst case, only one complete replica is required to survive the crash, and to hold the message. In such situation, the level of system reliability is the smallest, but its availability is uttermost, as the message obtained by the *RMU* replica is immediately passed to its recipient. In the proposed solution it is a role of a *Leader* to check safety of messages.

The idea of replicating the request among *RMU* replicas is the following: each time the *Interceptor* obtains the new request, it adds it to the log $\mathcal{M}$ (being a set of requests and replies exchanged between clients and services, and stored by each *RMU* replica), and broadcasts it to other replicas. The fact of delivering request to a replica is acknowledged by the communication channel, and results in adding such a *RMU* replica to the set $\mathcal{A}$ (a set of *RMU* replicas that acknowledged obtaining the request). The *Leader*, after obtaining request from *Interceptor*, can immediately send a reply to the client and to *Interceptor*, provided it possess the matching reply in its log $\mathcal{M}$. Otherwise, *Leader* is responsible for forwarding the request to *SIM* of the requested service. Since in the considered replication scheme the request is forwarded only after N requests replicas exist, the knowledge on the number of replicas maintained in the system is essential. For this purpose, each *RMU* replica, after obtaining a request and storing it in its log, sends acknowledment to *Leader* that updates its set $\mathcal{A}$. But, since the receipt of request can be acknowledged by some *RMU* replicas either to *Interceptor* or to *Leader* (for example in the case of communication channels with a low bandwidth between the *RMU* replica and *Leader*), *Interceptor* informs *Leader* , by sending its $\mathcal{A}$ to the *Leader*, which *RMU* replicas possess a request. In turn, *Leader* expands its set $\mathcal{A}$ on the basis of information obtained from *Interceptor.* Additionally, *Leader* broadcasts request to all *RMU* replicas that did not confirm obtaining this message neither to *Interceptor*, nor to *Leader.*

The purpose of the re-broadcast of request is to allow *RMU* replica that have just recovered from the failure to take part in the replication. In case one *RMU* replica acts as both the *Interceptor*, and the *Leader*, the procedure of updating the set $\mathcal{A}$ is simplified. There may be *RMU* replicas which do not belong to the set, although they possess the request message replica. However, this does not affect the correctness of the proposed solution. After performing the request, service provider returns the reply through its SIM to the *Leader*. When *Leader* obtains reply for the first time, it stores it in the log $\mathcal{M}$, and broadcast it to all *RMU* replicas to replicate reply along with its corresponding request. Finally, the *Leader* sends reply to the client. The architecture of replicated ReServE is shown in Fig.1.

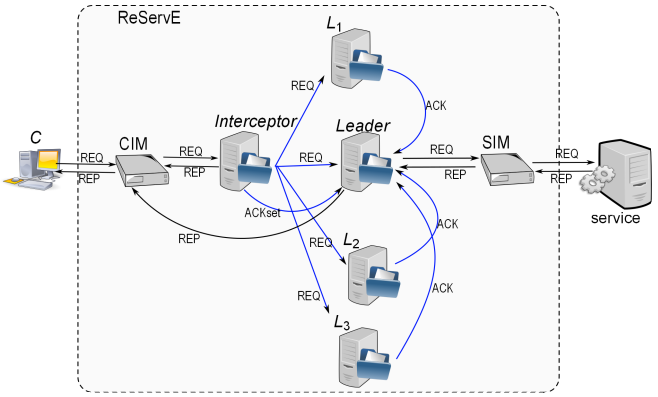

Fig. 1: ReServE — replication of recovery log

Despite reliable communication channels, an *Interceptor, Leader,* and other *RMU* replicas can fail, which effectively disturbs the communication between clients and service providers. In order to mask transient communication failures, the client reissues its request when no reply has been received within a given time. Thus, the role of the *RMU* replica is twofold: keep requests for the purpose of service recovery, and replies for the purpose of client recovery or for filtering duplicated request. In the case of the *Leader* crash another *RMU* replica must be elected to take the responsibility for further communication with the service. The *Interceptor* suspects the *Leader* crash in two cases. First, when the acknowledgment of obtaining a request broadcasted by the *Interceptor* is not delivered to *Leader*, and when the acknowledgment of obtaining by the *Leader* a request reissued by the client is not delivered. In both cases the *Interceptor* starts the *Leader* election procedure.

5 Conclusions and Future Work

In ReServE service considered until now, we have assumed that each service component can be the subject of failure, except of persistent storage of *RMU*,

where the message log is stored. Since this approach understates the robustness of message log, in this paper we proposed the preliminary concept of the alternative solution, based on the replication of *RMU* and its log.

Applying the replication mechanism will always introduce an overhead. But, the preliminary performance tests show that in the case of the specic message size, the overall costs of the proposed approach based on the replication are not inferior to the cost associated with the costs of saving messages in the stable storage. Thus, the proposed solution is competitive with the one based on persistent storage, because at similar costs, it increases recovery log availability.

Our future work encompasses the introduction of detailed protocol of recovery log replication, formal proof of its corectness, and detailed empirical evaluation.

References

1. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
2. Brzeziński, J., Danilecki, A., Hołenko, M., Kobusińska, A., Kobusiński, J., Zierhoffer, P.: D-reserve: Distributed reliable service environment. In: Morzy, T., Härder, T., Wrembel, R. (eds.) ADBIS 2012. LNCS, vol. 7503, pp. 71–84. Springer, Heidelberg (2012)
3. Brzeziński, J.: Dependability infrastructure for SOA applications. In: Ambroszkiewicz, S., Brzeziński, J., Cellary, W., Grzech, A., Zieliński, K. (eds.) Advanced SOA Tools and Applications. SCI, vol. 499, pp. 203–260. Springer, Heidelberg (1991)
4. Danilecki, A., Hołenko, M., Kobusińska, A., Szychowiak, M., Zierhoffer, P.: ReServE service: An approach to increase reliability in service oriented systems. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 244–256. Springer, Heidelberg (2011)
5. Elmootazbellah, N., Elnozahy, A.L.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3), 375–408 (2002)
6. OASIS. Reference Architecture Foundation for Service Oriented Architecture - Version 1.0 (October 2009)
7. Pedone, F., Wiesmann, M., Schiper, A., Kemme, B., Alonso, G.: Understanding replication in databases and distributed systems. In: Proceedings of the 20th International Conference on Distributed Computing Systems, Taipei, Taiwan, April 10-13, pp. 464–474 (2000)
8. Rao, S., Alvisi, L., Vin, H.M.: The cost of recovery in message logging protocols. In: SRDS, pp. 10–18 (1998)
9. Shahzad, F., Wittmann, M., Kreutzer, M., Zeiser, T., Hager, G., Wellein, G.: A survey of checkpoint/restart techniques on distributed memory systems. Parallel Processing Letters 23(4) (2013)