2012 IEEE 16th International Enterprise Distributed Object Computing Conference Workshops

An Illustrative Recovery Approach for Stateful Interaction Failure of Orchestrated Processes

Lei Wang[1], Andreas Wombacher[1], Luís Ferreira Pires[1], Marten J. van Sinderen[1] and Chihung Chi[2]
[1]Centre for Telematics and Information Technology, University of Twente
[2]School of Software, Tsinghua University
Contact email: {wangl, a.wombacher, l.ferreirapires, m.j.vansinderen}@ewi.utwente.nl, chihungchi@gmail.com

***Abstract*—During a stateful interaction, a partner service may become unavailable because of a server crash or a temporary network failure. Once the failed service becomes available again, the interaction partners do not have any knowledge about each other's state, possibly resulting in errors or deadlocks. This paper proposes an approach to the recovery of stateful interactions based on service interaction patterns and process transformations. Our recovery approach works without a central management node and without additional communication protocols. We also minimize the changes to the description of the service supported by the recovery-enabled process. Our approach allows one partner process to be modified in order to support failures in a way that interaction with the other (unchanged) processes is still possible.**

Keywords- Stateful Services; WS-BPEL; Failure Recovery

I. INTRODUCTION

During stateful interactions between business processes, a partner may fail because of a system crash or a network failure. Once the failed service becomes available again or the network connection is re-established, the interaction partners do not have any knowledge about each other's state, possibly resulting in errors or deadlocks. Recovery in this case often has to be performed manually after checking execution traces, which is potentially slow and expensive.

In this paper, we propose an approach to the recovery of stateful interactions of orchestrated processes. Our approach is based on interaction patterns and process transformations. The basic idea is to redesign the original processes into their recovery-enabled counterparts via process transformations that can be automated. We assume that in case a stateful process crashes, the state of each process instance has been persisted, so that the values of the process variables can be restored when the process restarts. This is a reasonable assumption since this behaviour is supported by most available WS-BPEL [1] engines, such as Apache ODE. Our recovery approach has been applied to WS-BPEL business processes, but it is general enough to be applicable to other process languages.

Fault handling approaches [2] require that the process designers are aware of possible failures and their recovery strategies. Alternatively (automated) process transformations can be defined to add generic recovery behaviours to orchestrations. Recovery mechanisms implemented as plug-ins for a WS-BPEL engine [3] strongly depend on a specific WS-BPEL engine. The approach to recovery presented in [4, 5] consists of replacing a failed service with another one. Transaction-based process recovery approaches [6, 7] require a central coordinator, in contrast with our approach, which is based on process transformations.

This paper is further structured as follows. Section II identifies interaction patterns that require recovery, Section III discusses the recovery approaches applied to each interaction pattern. Section IV gives an illustration to our approach. Finally, Section V concludes the paper.

II. STATEFUL INTERACTION FAILURE ANALYSIS

In order to determine which interactions between services require compensations in case of failures we considered the service interaction patterns as defined in [8]. From these patterns we initially considered the *send/receive* pattern, since it may incur state inconsistencies in case of failure. The failure analysis of the *send* and *receive* patterns can be ignored here, since they do not generate failure situations that are not covered by analyzing the *send/receive* pattern. The other interaction patterns will be investigated in future work. We assume that interactions are stateful. The process engine that gets a message routes this message to its corresponding instance based on the message contents.

The *send/receive* interaction pattern is shown in Fig. 1; Case I shows a synchronous interaction, while Case II shows an asynchronous interaction consisting of two one-way message exchanges. In the synchronous case, the initiator may fail after the request message ReqMsg is sent. We name this error pending request failure (denoted as X_{Req} in Fig. 1(a)). If the responder fails before message ReqMsg is received, or ReqMsg is lost because of a network failure, a failure occurs that we name service unavailable failure (denoted as X_{SU} in Fig. 1(a)). If the responder fails before the response message RespMsg is sent, or the response message RespMsg is lost because of network failure, a failure occurs that we name pending response failure (denoted as X_{Resp} in Fig. 1(a)).

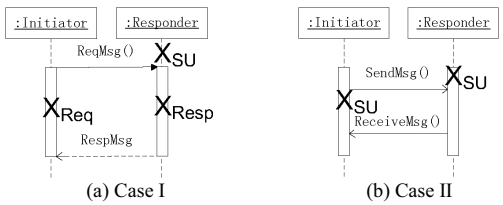


Figure 1. Interaction failure analysis: *Send* pattern.

In the asynchronous case, the partners may fail before receiving the notification message SendMsg or ReceiveMsg, or

978-0-7695-4786-2/12 $26.00 © 2012 IEEE
DOI 10.1109/EDOCW.2012.15

one of these messages is lost because of network failure. This also corresponds to the Service unavailable failure identified before (denoted as X_{SU} in Fig.1(b)).

III. RECOVERY APPROACH

Below we present the recovery solution we defined for each of the failures identified before.

A. Recovery Approach for Pending Request Failure

If the initiator process fails after sending the request, as is shown in Fig. 1(a), the responder process may still send the response and continue its execution.

1) Recovery Approach For Initiator Process: The initiator process should resend the request once recovered (a process engine restart or network connection re-establish).

2) Recovery Approach For Responder Process: The responder process needs to be transformed based on the assumption that there will be still further interaction between sender and receiver processes. As is shown in Fig. 2(a), based on the original process design, we add a while iteration with a pick branch in it. The while iteration is used for the processing of the multiple times of request resent by interaction initiator. On the left hand side of the pick branch, the activities receive1 and reply1 are used to response to the request resent from initiator. Note that this time the process will reply without processing. On the right hand side of the pick branch, the activity receive2 represents possible processes interaction. However, inorder to distinguish a new request from the one that has been re-sent, assumption has to be made that the two messages are of different types.

B. Recovery Approach for Pending Response Failure

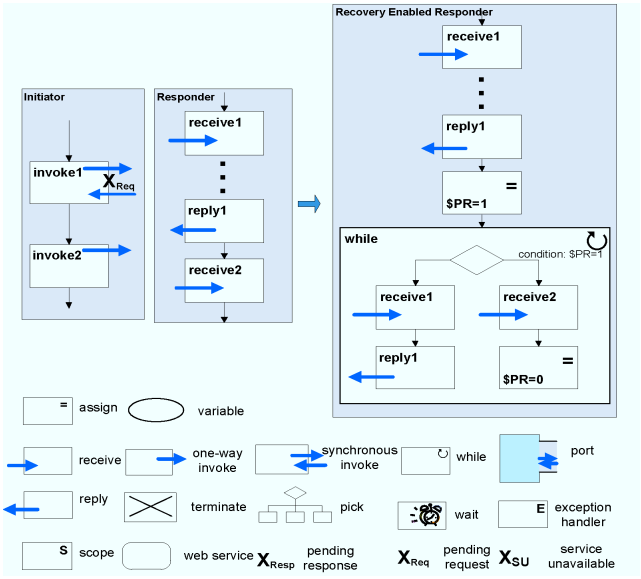
The responder process receives request, processes it using some activities nested between the request and the response. It then fails before sending the response. The failure will halt the execution of the responder process. The established connections for interactions will be lost.

1) Recovery Approach For Responder Process: In order to avoid the nested activities design, we split one synchronous interaction into two, as is shown in Fig. 2(b). One synchronous interaction (receive1 and reply1) is to send the parameters. The other (receive2 and reply2) is to return the response. Then the failure during the execution the nested interaction part will not interfere the execution of the initiator process because there is no open connections between the processes.

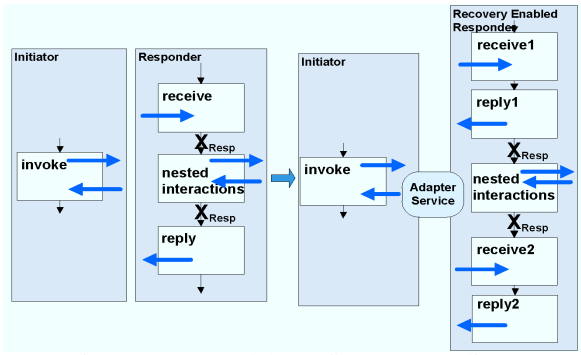
2) Recovery Approach For Initiator Process: With the adapter service used to keep the process interface, the adapter service receives the request from the initiator, interacts with responder and sends response back to initiator. Since the responder process may crash, the adapter service should be deployed together with the initiator.

C. Recovery Approach for Service Unavailable

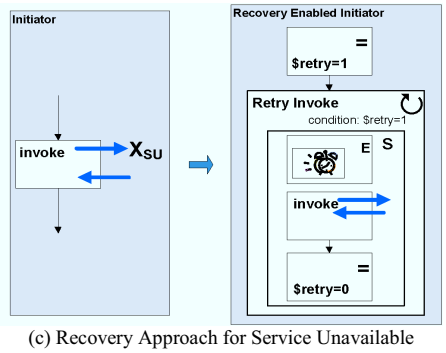
In order to deal with possible service unavailable failure, the recovery approach tries to transform the invoke activities in the following way. Inside a while iteration called Retry Invoke, we put a scope activity with a fault handler. When the target process service is not available, the fault handler will delay the execution of the process and the outside iteration will make the request sent again. However, race conditions between timeouts in initiating and responding processes may result in exceptions due to unexpected messages in the responding process. A pragmatic solution is to define proper timeout values for resending the invocation.



(a) Recovery approach for Pending Request Failure.



(b) Recovery approach for Pending Response Failure.



(c) Recovery Approach for Service Unavailable

Figure 2. Process transformation approach.

IV. EXAMPLE SCENARIO

We illustrate our approach with a scenario shown in Fig. 3 of a simple procurement process in a virtual enterprise. It contains three business partners: a buyer, an accounting department and a logistics department. The accounting department gets a *getQuote* message from the buyer and returns a *quote* message. The *order* information (*deliver* message) is forwarded by the accounting department to the logistic department. The logistic department then confirms the receipt (*deliverConf* message with expected delivery date and parcel tracking number) to the accounting department. The accounting department forwards a *delivery* message to the buyer. Furthermore, the buyer can decide to track the status or terminate the process (messages *getStatus*, *status*, or *terminate*). All these interactions are stateful because the execution engine of each participant needs to route each received message to its corresponding instance.

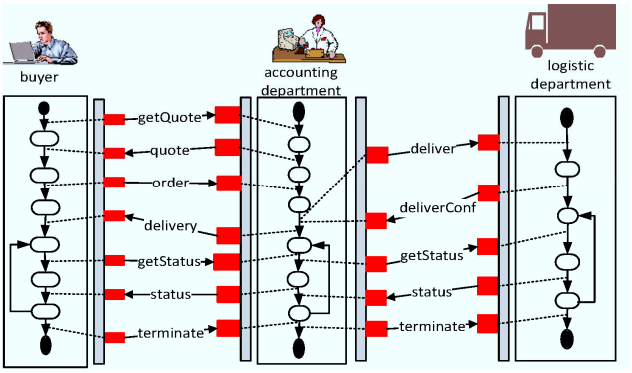


Figure 3. Overview of the example scenario.

The BPEL definition of the accounting department process is shown in Fig. 4.

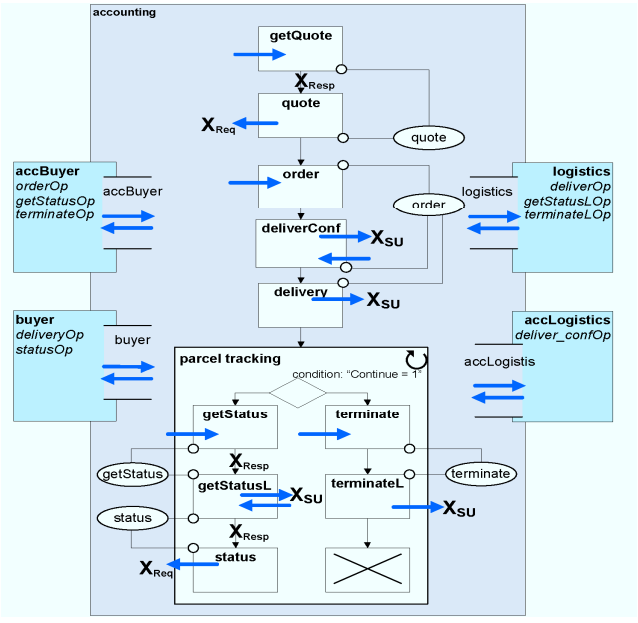


Figure 4. Accounting BPEL process.

The scenario starts by receiving a synchronous invocation *getQuote* message requiring the quote information. After the accounting process replies with a *quote* message, the buyer sends the *order* message, which is forwarded to the logistic process by the accounting process. The logistic process then replies with a *deliverConf* message to the accounting process. The message is forwarded to the buyer via a *delivery* message afterwards. Since the buyer is allowed to do parcel tracking arbitrarily often, this step is embedded in a *while* iteration within the accounting process. More precisely, the accounting department may receive a *getStatus* message sent by the buyer, which is then followed by a synchronous invocation of the logistics *getStatusLOP* operation and the reply of the respective status back to the buyer (via a status message). Alternatively, the buyer may decide terminate the accounting and the logistics process at some point by sending a *termination* message to the accounting process, which is forwarded to the logistic process.

A. Orchestrated Processes Interaction Failure Analysis

All possible failure points are depicted out in Fig. 4. The process begins with a *getQuote* receive activity and a *quote* reply activity. If the process crashes after receiving the request message *getQuote* and before sending a response, this should be a pending response failure, which is marked as a X_{Resp} between the *getQuote* and *quote* activities. For similar reasons, in the *parcel tracking* while iteration, if the accounting process crashes after the receive activity *getStatus* and before the *status* reply, a pending response failure will occur. We can notice that there is a nested invoke activity in between.

For the *quote* reply activity, if the buyer process which should receive this reply fails after sending the request, a pending request failure will happen to the accounting orchestration, which is marked as a X_{Req}. A pending request failure will happen at the point of *status* reply for the same reason. If the process crashes after the *quote* activity and before receiving the *order* information, the buyer process that sends the order information will have a service unavailable failure. However, since the failure happens at the buyer side, we don't need to transform the accounting process for any recovery. The order information exchange is already finished. For the same reason, the *terminate* activity is not marked with any failure if the process crashes before or after this activity. For the invoke activity *deliverConf*, which invokes the *deliverOP* operation provided by the logistics process, if the logistics process is not available before the invocation, a service unavailable failure will happen to the accounting process, which is marked as a X_{SU}. For similar reasons, a service unavailable failure will happen to activities: *delivery*, *getStatusL* and *terminateL*.

B. Accounting Process Transformation

As is shown in Fig. 5, in order make the accounting process recovery enabled, we apply all necessary process transformations. For the reply activity *quote* where a pending request failure may occur, we do the transformation to put a conditional branch (*pick* activity) in a *while* iteration. If the buyer process crashes and resends the *getQuote* message, the accounting process will reply with *quote* information in the *while* iteration, until *order* information is received.

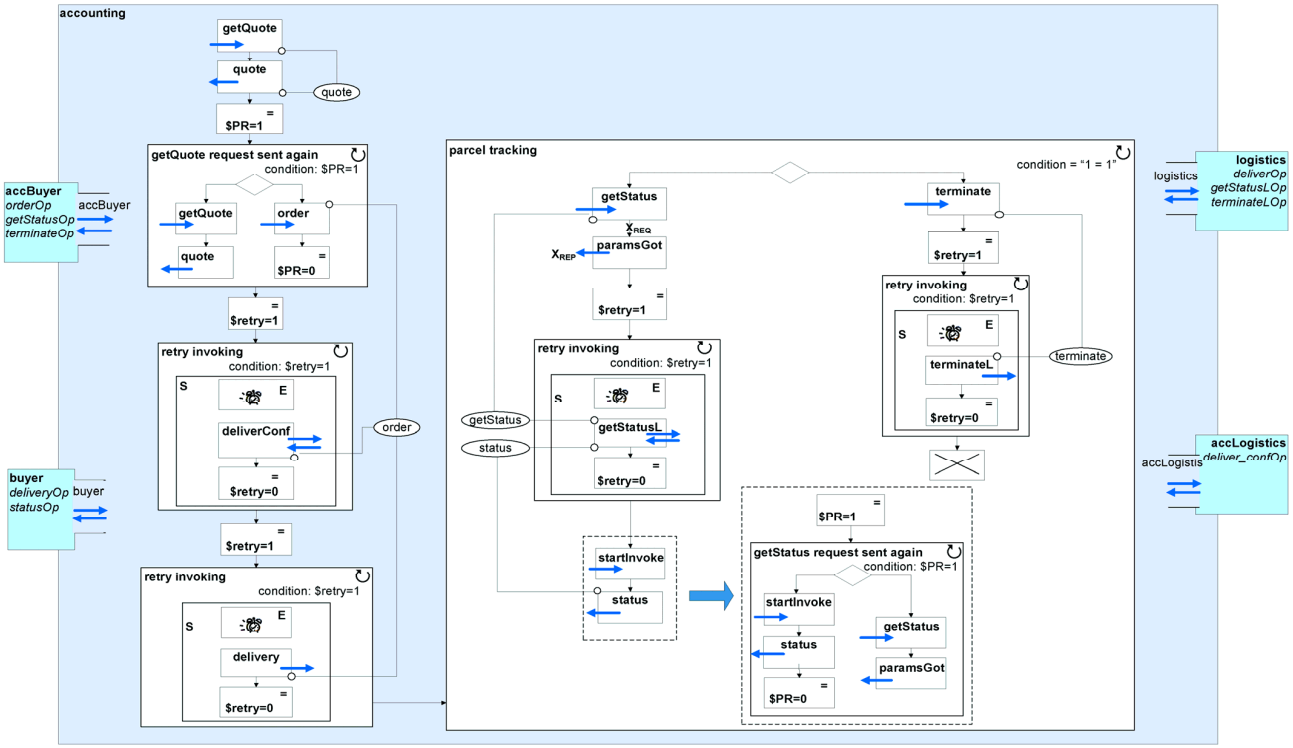


Figure 5. Transform Accounting Process to Make it Recovery Enabled

For pending response failures (marked as X_{Resp}) that occur between *getStatus* and *status*, in order to avoid a nested *getStatusL*, we make the parameters of *getStatus* request sent with an immediate response. After the interactions with other processes, the buyer process may send the request again in order to get the *status* reply. However, after this transformation, as is shown in Fig. 5, we have two new points of failure which need further transformations to make these failures recoverable. The transformations are shown in the dotted rectangle.

V. CONCLUSIONS

In this paper, we propose a recovery approach for stateful interactions between processes. We perform a stateful interactions failure analysis for orchestrated processes and we propose a recovery approach for each failure we identified. We also define transformations from the original processes to processes that support our recovery approaches. Finally, we illustrate our approach with a simple yet representative scenario. We have implemented the illustration scenario using BPEL on the Apache ODE BPEL engine. Our next challenge is to evaluate and probably improve the scalability of the recovery-enable BPEL processes. A formal proof of the correctness of process transformations will be considered as future work, as well as the automation of these transformations, for example, by using model transformations from Model-driven Engineering (MDE).

REFERENCES

[1] OASIS, "Web Services Business Process Execution Language Version 2.0," http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[2] N. Russell, W. M. P. van der Aalst, and Arthur, "Exception Handling Patterns in Process-Aware Information Systems," BPMcenter.org, Tech. Rep., 2006.

[3] S. Modafferi, E. Mussi, and B. Pernici, "Sh-bpel: a self-healing plug-in for ws-bpel engines," in *the 1st workshop on Middleware for Service Oriented Computing*, ser. MW4SOC '06. ACM, 2006, pp. 48–53.

[4] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras, "Dynamic service substitution in service-oriented architectures," in *Proceedings of the 2008 IEEE Congress on Services - Part I*, july 2008, pp. 101 –104.

[5] L. Cavallaro, E. Di Nitto, and M. Pradella, "An automatic approach to enable replacement of conversational services," in *Service-Oriented Computing,* ser. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2009, vol. 5900, pp. 159–174.

[6] OASIS, "Web Services Atomic Transaction Version 1.2," WWW page, 2009. http://docs.oasis-open.org/ws-tx/wsat/2006/06

[7] OASIS, "Web Services Business Activity Version 1.2," WWW page, 2009. http://docs.oasis-open.org/ws-tx/wsba/2006/06

[8] A. Barros, M. Dumas, and A. ter Hofstede, "Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection," Technical Report FIT-TR-2005-02, Faculty of Information Technology, Queensland University of Technology, 2005.