A Collaborative Processes Synchronization Method with Regards to System Crashes and Network Failures

Lei Wang*, Andreas Wombacher*+, Luís Ferreira Pires*,
Marten J. van Sinderen*, Chi-Hung Chi†
*Centre for Telematics and Information Technology, University of Twente
†Computational Informatics - Hobart, CSIRO, Tasmania, Australia
+Nspyre, The Netherlands
wangl@ewi.utwente.nl, {a.wombacher, l.ferreirapires, m.j.vansinderen}@utwente.nl,
chihungchi@gmail.com

ABSTRACT

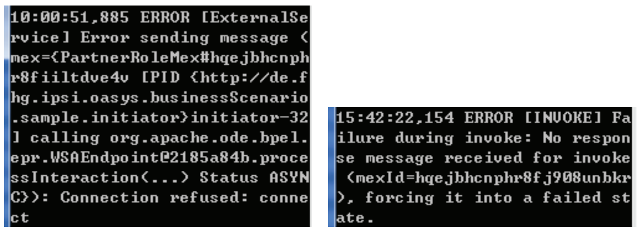
Processes can synchronize their states by exchanging messages. System crashes and network failures may cause message loss, so that state changes of a process may remain unnoticed by its partner processes, resulting in state inconsistency or deadlocks. In this paper we define a method to transform a business process into its recovery-enabled counterpart. We also discuss the correctness proof of the transformation, and the performance evaluation of our prototype implementation. In our previous work, we presented solutions to these synchronization problems that were based on rather strong assumptions. For example, specific failure patterns or interaction patterns (one client instance interacts with one server instance) were assumed. In this paper, the solution is extended to multiple process instances with more possible synchronization failures.

Keywords

Robust, Recovery, State Synchronization, Service Interaction, Business Process, WS-BPEL

1. INTRODUCTION

Electronic data interchange has grown significantly in the last decade. Often data interchange is based on collaborative processes run by different parties exchanging messages to coordinate the execution of their business processes. At runtime, each process instance maintains its state. The state change of one party is synchronized with its partners via messages exchange. A message loss may happen due to the possibility of system crashes and network failures,thus a state change by one party may remain unnoticed by the other parties. This may result in inconsistent states or even dead-locks. Thus robust state synchronization mechanisms are necessary to guarantee proper collaborative process execution. Possible synchronization failures can be seen from a screen dump of an error after a system crash of an orchestration engine such as Apache ODE (see Fig. 1)

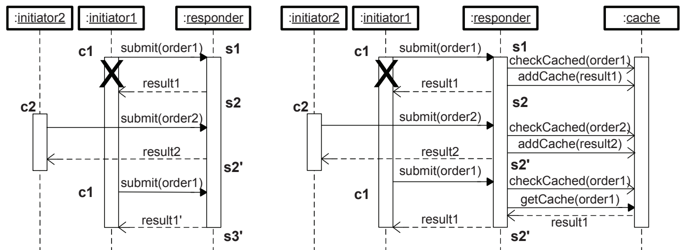


```
10:00:51,885 ERROR [ExternalSe
rvice] Error sending message (
mex={PartnerRoleMex#hqejbhcnph
r8fiiltdve4v [PID {http://de.f
hg.ipsi.oasys.businessScenario
.sample.initiator}initiator-32
] calling org.apache.ode.bpel.
epr.WSAEndpoint@2185a84b.proce
ssInteraction(...) Status ASYN
C}): Connection refused: conne
ct
```

```
15:42:22,154 ERROR [INVOKE] Fa
ilure during invoke: No respon
se message received for invoke
 (mexId=hqejbhcnphr8fj908unbkr
), forcing it into a failed st
ate.
```

(a) Service Unavailable (b) Pending Response

Figure 1: Synchronization Failures



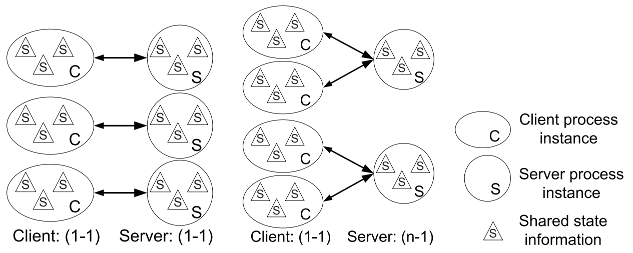
(a) A possible failure (b) Idea: cache based solution

Figure 2: Our Idea

A possible failure situation is illustrated in Fig. 2a) using a simple purchase process. In the purchase process, the *initiator1* submits the order, and we assume that a system crash occurs afterwards. During the failure of *initiator1*, *responder* sends its result message and changes into state $s2$. The *responder* further changes from state $s2$ to $s2'$ due to a synchronization with *initiator2* also submitting an order. A request is said to be "idempotent" [15] if the operation can be safely repeated. However, the message $submit(order)$ is not idempotent. We can conclude this because the responder changes its state from $s1$ to $s2$ after receiving message $submit(order)$. If the *initiator1* recovers simply by resending the *submit(order)* message, which as a consequence triggers further *responder* state change from $s2'$ to $s3'$ (possible order re-processing) and a *result'* message, which most probably will be different from the original response *result1*.

A *private* interaction pattern [1] (see Fig. 3a) represents

http://dx.doi.org/10.1145/2554850.2554980



(a) private (b) shared, static (c) legend

Figure 3: Multiple Processes Instances Shared State Types

a state synchronization between a pair of client and server instances. While a *shared, static* interaction (see Fig. 3b) means that state is shared between multiple client instances and static number of server instances. We present in [8] a method to determine non-idempotent operations. In this paper we propose an approach addressing the more complex *shared, static* interaction with non-idempotent operations. The basic idea (see Fig. 2b) behind it is to cache the response of non-idempotent operations. In case a request is resent in the context of a recovery action, the request is not executed again and the cached response is returned. In our previous work, we presented solutions to these problems that were based on rather strong assumptions. In [16], a *private* interaction pattern is assumed. In [8] we extend our solution to cover the *shared, static* interaction pattern, however, only one possible synchronization failure is considered. In this paper, the solution is extended to cope with the more complex *shared, static* interaction pattern with more types of synchronization failures.

We assume that in the case of server crashes or network failures, the state of the business process can be restored once recovered. This is a reasonable assumption, since most available business process engines, such as Apache ODE, work in this way. We choose WS-BPEL [12] as an illustrative process specification language, because as an OASIS standard it is widely used by enterprises. However, our mechanisms are applicable to other process specification languages that support similar workflow patterns [17]. The structure of the paper is the following. Section 2 analyzes possible synchronization failures. Section 3 proposes our process transformation based solution. Section 4 evaluates our solution. Section 5 discusses related work and Section 6 concludes our paper.

2. SYNCHRONIZATION FAILURE ANALYSIS

Possible synchronization failures with regards to system crash and network failure are *service unavailable*, *pending request failure* and *pending response failure* [16]. All these synchronization failures are shown in Fig. 4. A *service unavailable failure* (marked as X_{SU}) happens if the responder system crashes or the network fails before a request message is delivered. A *pending request failure* (marked as X_{REQ}) happens if the initiator crashes after sending the request and before receiving the response message. A *pending response failure* (marked as X_{RESP}) happens if the responder system crashes after receiving the request but without sending the response message. Another reason for a pending response failure is that the network fails when delivering the response message.

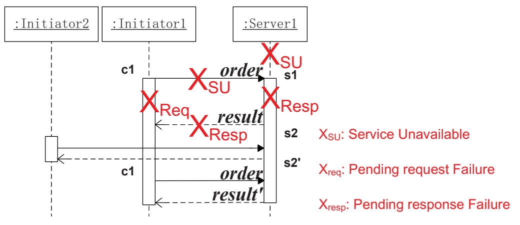

:Initiator2
:Initiator1
:Server1
c1
order
result
order'
result'
s1
s2
s2'
X SU
X Req
X Resp
X SU: Service Unavailable
X req: Pending request Failure
X resp: Pending response Failure

Figure 4: Possible Synchronization Error

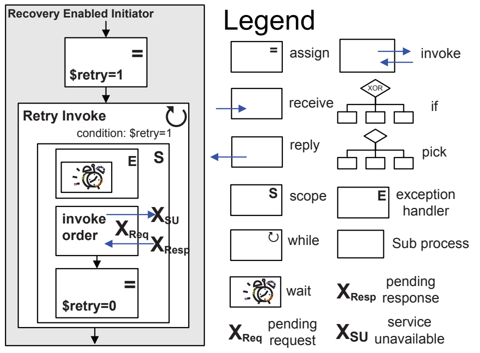


Figure 5: Transformed Initiator Process

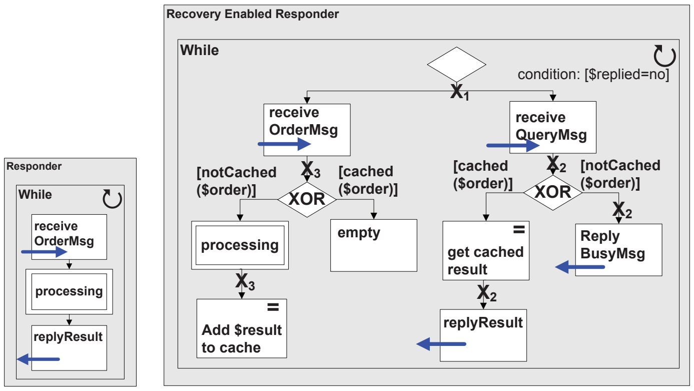
The difficulty of recovery is twofold. First, a responder state change from *s1* to *s2* is triggered by a synchronization (*order* message submission from single client). Second, because multiple *initiator* instances (*initiator*1, *initiator*2) synchronize their states with one *responder* instance, if *pending request failure* happens, further state changes are still possible. The server instance may not stay in state $s2$ and may change into a state $s2'$ due to synchronization with another initiator instance (*initiator*1).

3. RECOVERY METHOD

Our idea is to transform the original process into a recovery-enabled process. On the initiator side, the main idea of the transformation is to resend the request message whenever a synchronization failure is detected. On the responder side, the transformation adds a caching capability, i.e., the response message for a newly incoming message representing a non-idempotent operation is cached. If the responder receives a resent message from the initiator due to a failure, the responder replies the cached response message and does not execute the operation again. In the following we discuss the initiator and responder transformations.

3.1 Initiator Transformation

The initiator starts a state synchronization by executing an *invoke* activity. The transformation of the *invoke* activity is shown as Fig. 5. The *invoke* activity is nested within a *scope* activity with an exception handler. If a synchronization failure happens (marked as X_{SU}, X_{Req}, X_{Resp}), the exception handler in the scope can be executed, i.e., adding a delay before retrying using a *wait* activity. A failure is detectable in many WS-BPEL process engines, e.g., Apache ODE. The *scope* activity is inside a *while* activity to imple-



(a) original (b) transformed

Figure 6: Transformation of the Responder Process

ment a retry behavior. If the *invoke* activity finishes without failure, an *assign* activity is used to alter the value of the variable $\$retry$ to 0 to end the while iteration.

3.2 Responder Transformation

The original responder process is shown in Fig. 6a. The process receives an order message, processes it, and sends a response. The transformed responder process is shown as Fig. 6b. The state synchronization consisting of a request response pattern is split into an asynchronous request to send the original order message, and a request response pattern to query the response, thus the result of the original request.

In the first step, the responder receives a message $OrderMsg$ (the left branch of the *pick* activity in the *while* iteration in Fig. 6b). This is a one-way message, so that the responder does not send a response, thus pending request or response failure are avoided. After receiving the order message, an *if* activity is used to check whether the order is cached. If the order is cached, this implies that the order has been processed before and this is a resent message due to failure. In this case the responder does nothing (*empty* activity). If the order is not cached, the responder processes the order and adds the result to the cache.

In the second step, the responder receives the query message from the initiator. The WS-BPEL correlation mechanism can be used to correlate the query message with the corresponding order message. If the order is cached, the responder will use the cached response message as a reply. If the request is not cached, the responder sends a message $BusyMsg$ to indicate to the initiator that the processing is not finished.

The two steps are placed via a pick activity in a while iteration to support the interaction with multiple client instances and retries per instance.

After the transformation, the possible types of failures are marked as X_1, X_2 and X_3. Failure X_1 is a service unavailable failure and can be compensated by the initiator's transformation support to resend a message. Failure X_2 is a pending response failure. The initiator can detect the failure by not receiving the response message and recover by resending the query message. On the responder side, the resend message is replied with a cached response. Failure X_3 happens in a control flow outside a synchronization block, thus, this error does not affect the state synchronization with the initiator.

3.2.1 Implementation of Cache Related Operations

The cache is declared as a process variable in WS-BPEL with an XML structure of entries. Each entry is a mapping from request message to response message. A sample cache entry is shown as the following.

```
<cache>
 <cacheEntry>
  <requestEntry>
   <requestMsg />
  </requestEntry>
  <responseEntry>
   <responseMsg />
  </responseEntry>
 </cacheEntry>
 <cacheEntry .../>
</cache>
```

The cache operations are pre-defined using XSLT to operate on the cached XML data. The invocation of cache operation is an "assign" activity. We use the standard WS-BPEL function $doXslTransform()$ in the assign with a predefined XSLT script to manage the cache. A sample read cache activity is shown as the following.

```
<bpel:assign>
 <bpel:copy>
  <bpel:from>
   bpel:doXslTransform(
   "testCached.xsl", $cache,
   "requestMsg", $requestMsg)
  </bpel:from>
  <bpel:to>$foundCachedRequest</bpel:to>
 </bpel:copy>
</bpel:assign>
```

3.2.2 Adapter Design

As the transformation defined so far, there is a mismatch in the interaction patterns expected by the transformed responder and the transformed initiator. To solve this problem, the initiator is further transformed. In case it is not possible to modify the initiator, we have to place an adapter between the initiator and the responder to mediate this mismatch.

The design of the adapter process is shown in Fig. 7. The adapter receives an *order* message from the initiator. In the following *while* iteration, it forwards the message to the responder (activity *invoke1*). Then it sends the query message to ask the responder for the result (activity *invoke2*). If the *result* message is replied ([$\$reply = result$]), the *while* iteration is terminated by changing the value of the conditional variable $\$finish$ to *yes*. If the result message is $MBusy$, which indicates that the responder is still processing, a *wait* activity is executed to introduce a delay and then the outer *while* iteration sends the *query* message again. Finally, the *result* message is replied to the initiator. Both *invoke* activities are defined in the while iteration, because the first *invoke* activity is a one-way message exchange, which is non-blocking. If the first *invoke* activity is defined outside the while iteration, it is possible that the transport layer delivers the second message (query) before the first message (order).

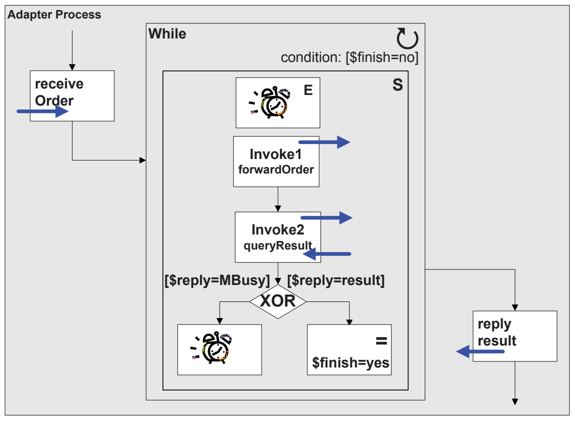


Figure 7: Adapter Process Design

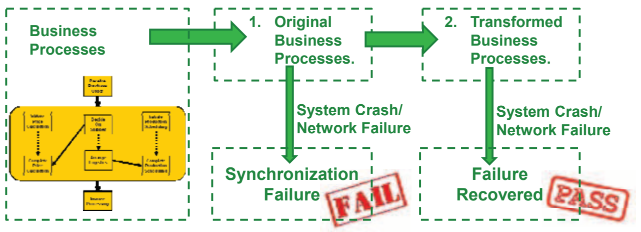

Business Processes
1. Original Business Processes.
2. Transformed Business Processes.
System Crash/ Network Failure
System Crash/ Network Failure
Synchronization Failure
FAIL
Failure Recovered
PASS

Figure 8: Correctness Evaluation Setup

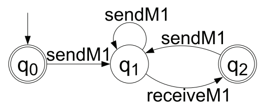


Figure 9: Correctness Criteria: One Way Message

From the initiator point of view, the adapter is designed as a stateless process: each client request triggers a new adapter creation. In the case of an adapter failure, this is a pending response failure from the client point of view. This triggers the client to resend the request message, which creates a new adapter instance to fulfill the synchronization.

3.2.3 *Design Considerations of Generic Adapter*

If we deploy a separate adapter for each specific initiator and responder, lots of adapter instances are created, which will probably increase the processing overhead. Another option is to re-use a generic adapter for all initiators and responders. Three related design considerations are discussed in the following. First, the messages delivered to and from adapters should be independent from the initiator and responder processes. The parts of the message should refer to a generic typed element or declared as a generic type, such as “xsd:anyType”. The drawback is due to the correlation mechanism of WS-BPEL. In particular, for different messages different correlation set configurations are required, which makes it necessary to distinguish messages, which is not the case with anyType. Second, the responder process should describe its operations in a process-independent way. The generic adapter should not refer to any responder specific operations, e.g., process specific port type definitions in their WSDL. The drawback is that the responder cannot use the control flow branching activities (like “pick” in WS-BPEL), because all messages are generic types and dedicated to generic process operations. Finally, a generic addressing mechanism is required to forward an incoming message to a proper responder, for example, by mapping specific information of an incoming message to the address of the responder. This can be achieved by using a mapping from message to responder addresses in XSLT. In WS-BPEL, the function doXslTransform(inMsg, XSLT) can be used to query the responder address.

From the above discussion, we can see that the possibility of using a generic adapter is quite limited. On the other hand, with additional process management effort, a process-specific adapter can still be automatically generated from the initiator and responder services descriptions, e.g., their WSDL descriptions.

4. EVALUATION

We evaluate three aspect of our recovery mechanism: the correctness of the process transformation, its performance at runtime and the process design complexity.

4.1 Correctness Evaluation

Fig. 8 shows that the correctness proof is done in the following three steps. First, we prove that a process interaction as shown in Fig. 6 is correct without system crashes or network failures. Second, we prove that the process interaction cannot pass the correctness criteria when a synchronization failure happens. Finally, with the process transformation method described in this paper, we prove that the process interaction can finish successfully with regards to synchronization failures.

4.1.1 *Correctness Criteria*

In this paper, correctness means that in the presence of a system crash or network failure, all the synchronization can be recovered. After a few retry messages are sent, all the messages can be ultimately received. Formally, we model the criteria of correctness using automata defined as tuple $A = (Q, \Sigma, \delta, q_0, F)$. A criteria automata for a synchronization using a one-way message M1 is shown in Fig. 9. The state set Q contains all possible states. q_0 is the initial state, q_1 is the state after M1 is sent and q_2 is the state after the message is finally received. The transition set Σ is the message sending or receiving behavior $\{sendM1, receiveM1\}$. F is the set of all acceptable states that we use to describe that the synchronization finished successfully $\{q_0, q_2\}$. The transition function is described as follows:
$\delta(q_0, sendM1) = q_1$: The message $M1$ is sent. $\delta(q_1, sendM1) = q_1$: If the message is not received, it may be re-sent multiple times. $\delta(q_1, receiveM1) = q_2$: The sent message finally got received. $\delta(q_2, sendM1) = q_1$: Message of the same type is sent multiple times from sender. This is possible due to specific process design: the send message activity could be nested in an iteration (while loop). An additional criteria model for synchronous interactions has been described in our previous work [8].

4.1.2 *Evaluation Process*

In the evaluation, we model the WS-BPEL using Petri Nets. In our previous work [8], a mapping of WS-BPEL activities to a Petri Net model is specified. By simulating the Petri Net model, we generate the corresponding occur-

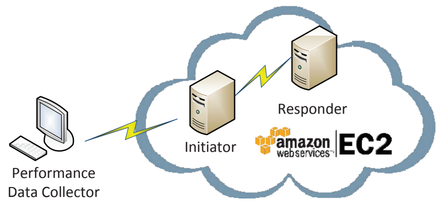


Figure 10: Setup of Performance Test

Table 1: Performance of Our Transformation

Workload	Before Transformation	After
$\lambda = 5$	645 ms	1607 ms
$\lambda = 10$	892ms	5419 ms

rence (automata) graph. Finally we prove the correctness by proving that the generated automata is subsumed by the criteria automata. The detailed proof is omitted here due to page space limitations.

4.2 Performance Evaluation

The setup of our performance test is shown in Fig. 10. We use the cloud infrastructure from Amazon EC2. The initiator and responder processes are deployed on two computing instances and we use a local client to collect the performance data. We test the performance of the original process and the transformed process under two different workloads. The request sent per minute complies to a Poisson distribution with parameters $\lambda = 5$ and $\lambda = 10$. Each test lasts for an hour and the response times of the executions of the 30 minutes in the middle are collected. The performance data is shown in Table 1.

Some performance overhead is caused by the set of process-specific query intervals in the adapter process. After the responder finishes processing it can only send the reply after the adapter has queried the result. For example, in our test, the query interval is 1000ms and we see the response time distribution in Fig. 11. The response time peak interval is near 1000ms, which is the query interval. Another reason could be the management of the adapter instances. At runtime, each incoming request message triggers a new adapter instance creation. As we use the limited EC2 instance type in this evaluation (*t1.micro* with 1 vCpu and 0.613GiB memory), we expect better performance under scalable infrastructure, e.g., the auto scaling feature of cloud.

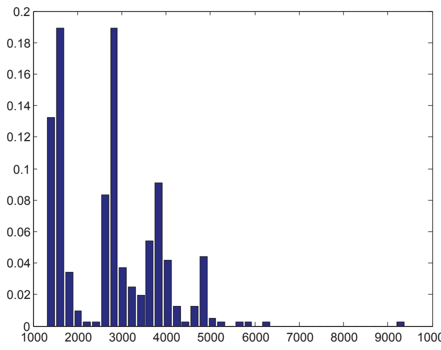


Figure 11: Response Time Distribution

4.3 Transformation Complexity Evaluation

The illustrative responder process depicted in Fig. 6a contains one *receive* and *reply* activity nested in a *while* iteration to process the request messages from multiple clients. The receive, process and reply activities are wrapped as a sequence activity. Thus in total 4 activities are defined. The pseudo-code of the corresponding transformed process is shown as follows.

```
<bpel:while>
 <bpel:condition>... </bpel:condition>
 <bpel:pick>
  <bpel:onMessage name="receiveParams" .>
   <bpel:if name="isReqMsgCached">
    <bpel:condition .></bpel:condition>
    <bpel:sequence>
     <!-- Some Process -->
     <bpel:assign name="cacheResult" .>
     </bpel:assign>
    </bpel:sequence>
   </bpel:if>
  </bpel:onMessage>
  <bpel:onMessage name="getResult">
   <bpel:if>
    <!-- Result is not added to cache-->
    <bpel:condition>...</bpel:condition>
    <bpel:reply variable="MBusy" . />
    <!--Result is in cache.-->
    <bpel:else>
     <bpel:assign name="getCachedReply"/>
     <bpel:reply name="replyResult" />
    </bpel:else>
   </bpel:if>
  </bpel:onMessage>
 </bpel:pick>
</bpel:while>
```

Without introducing the process design details we can notice the complexity of the transformed process. In total 9 activities are defined. However, in our approach this complexity is not revealed to process designers, since the process can be transformed automatically into its recovery-enabled counterpart during deployment.

5. RELATED WORK

Exception handling [2, 14] is process-specific. WS-BPEL supports compensations of well-defined exceptions using exception handlers. However, elaborate process handler design requires process-specific knowledge of failure types and their related recover strategies. Alternatively, we try to ease the process designers from dealing with synchronization failures by a transparent process transformation from a given business process to its recovery-enabled counterpart. If a system is to be fault tolerant, it can better try to hide the occurrence of failures by applying redundancy [15]. Three kinds of redundancy are possible: information redundancy, time redundancy, and physical redundancy.

Fig. 12 shows that physical redundancy is a recovery ability of the infrastructure. The solutions discussed in [10, 9, 3] are implemented as process engine plug-in, which makes them dependent of a specific process engine. We defined our solution based on the WS-BPEL building blocks without requiring extensions at engine level. The transformed process

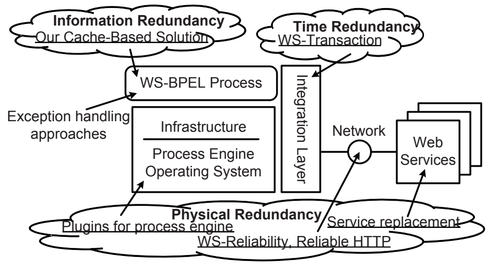


Figure 12: Related Work

can still be migrated to other standard process engines. Reliable network protocols such as HTTPR and WS-RX are proposed to provide reliable synchronization. However, the deployment of these solutions increase the complexity of the network infrastructure. We assume that system crashes and network failures are rare events, thus extending the infrastructure introduces too much overhead. Further, the solutions are not applicable in some out sourced deployment environment. For example, in some cloud computing environments, user-specific infrastructure configuration to enhance synchronization is not possible. Dynamic service substitution [6, 7] is a way to perform recovery by replacing the target services by equivalent services. In [11, 5, 4], the QoS aspects of dynamic service substitution are considered. In our work, we do not change the business partners at run time.

Information redundancy recovery is based on replication. Our cached-based process transformation is information redundant because a cache is a kind of replication. Time redundancy solutions include web services transactions. The WS-AT standard specifis atomic transactions, while WS-BA standard specifis a relaxed transactions that the participant can choose to leave the transaction before it commits. However, if a transaction rolls back, a process-specific compensation is required. Actually, transactions can deal with well-defined failures. The 2 phase commit distributed transaction protocol can not deal with system crash (referred to as *cite failure* in [13]). In a special case of process in which that all participants send the vote result to a coordinator, if the coordinator crashes before sending the vote result to any participant, all the participants are blocked without knowing the final results of the transaction.

6. CONCLUSIONS

In this paper, we propose a process synchronization failure recovery method with regards to system crashes and network failures. We analyze possible synchronization failures and we propose a recovery method to transform a business process into its recovery-enabled counterpart. We have proved the correctness of our process transformations and we implemented a prototype to test the runtime performance of our method. In future, synchronization failures of more complex process interactions patterns should be taken into consideration.

7. REFERENCES

[1] C. Atkinson and P. Bostan. Towards a client-oriented model of types and states in service-oriented development. In *IEEE 13th EDOC Conference*, pages 119–127, 2009.

[2] B. Staudt, et al. Exception handling patterns for process modeling. *IEEE Transactions on Software Engineering*, 36(2):162–183, 2010.

[3] A. Charfi, T. Dinkelaker, and M. Mezini. A plug-in architecture for self-adaptive web service compositions. In *IEEE Intl. Conf. on Web Services*, pages 35–42, 2009.

[4] F. Moo-Mena, et al. Defining a self-healing qos-based infrastructure for web services applications. In *11th IEEE Intl. Conf. on Computational Science and Engineering Workshops*, pages 215 –220, 2008.

[5] F. Moo-Mena, et al. A diagnosis module based on statistic and qos techniques for self-healing architectures supporting ws based applications. In *Intl. Conf. on CyberC.*, pages 163 –169, 2009.

[6] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras. Dynamic service substitution in service-oriented architectures. In *IEEE Congress on Services - Part I*, pages 101–104, 2008.

[7] L. Cavallaro, et al. An automatic approach to enable replacement of conversational services. In *Service-Oriented Computing*, volume 5900, pages 159–174. Springe Berlin Heidelberg, 2009.

[8] L. Wang et al. Robust client/server shared state interactions of collaborative process with system crash and network failures. In *10th IEEE Intl. Conference on Services Computing (SCC)*, 2013.

[9] S. Modafferi and E. Conforti. Methods for enabling recovery actions in ws-bpel. In *On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*, volume 4275, pages 219–236. Springer Berlin Heidelberg, 2006.

[10] S. Modafferi, E. Mussi, and B. Pernici. Sh-bpel: a self-healing plug-in for ws-bpel engines. In *the 1st workshop on M4SOC*, pages 48–53, NY, USA, 2006. ACM.

[11] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *the 17th Intl. Conf. on WWW*, pages 815–824. ACM, 2008.

[12] OASIS, `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`. *Web Services Business Process Execution Language*, 2.0 edition, Apr. 2007.

[13] M. T. Ozsu. *Principles of Distributed Database Systems*, chapter 12. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[14] N. Russell, W. Aalst, and A. Hofstede. Workflow exception patterns. In *Advanced Information Systems Engineering*, volume 4001, pages 288–302. Springer Berlin Heidelberg, 2006.

[15] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*, chapter 8. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 2006.

[16] L. Wang, A. Wombacher, L. Ferreira Pires, M. J. van Sinderen, and C. Chi. A state synchronization mechanism for orchestrated processes. In *IEEE 16th Intl. Enterprise Distributed Object Computing Conference (EDOC)*, pages 51–60, 2012.

[17] W.M.P. van der Aalst et al. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, Jul. 2003.