Providing Fault-tolerant Execution of Web-service–based Workflows within Clouds [*]

Johannes Behl,
Tobias Distler,
Florian Heisig
Friedrich–Alexander University
Erlangen–Nuremberg
{behl,distler,heisig}@cs.fau.de

Rüdiger Kapitza
TU Braunschweig
rrkapitz@ibr.cs.tu-bs.de

Matthias Schunter
IBM Research – Zurich
mts@zurich.ibm.com

ABSTRACT

With a variety of services rapidly evolving at all architectural levels of cloud computing, there is an increasing demand for a standardized way to coordinate their interactions. Business process management, that is, more general, the management of Web-service–based workflows, could satisfy this demand and, indeed, first corresponding offerings have gained instant popularity. While from a functional perspective, these Platform-as-a-Service (PaaS) solutions are already quite mature, their support for fault tolerance is still very limited, making them inapplicable for critical tasks.

Concerning the deficiencies of currently existing systems, this paper presents a practical solution for executing critical Web-service–based workflows, particularly within clouds, in a fault-tolerant, highly available and highly configurable manner. We achieve this by actively replicating workflows as well as Web services in a combined architecture, reusing existing standard systems and coordination services. By providing an automated transformation tool, replication is realized transparently to existing systems and workflows. Measurements show that our proposed architecture achieves lower response times than existing systems and that the integration of a coordination service imposes only moderate costs, while simplifying the implementation and leading to a dynamically adaptable solution.

1. INTRODUCTION

Following the slogan "Everything as a service", cloud computing leads to a constantly growing number of services, ranging from very basic infrastructure offerings such as computing power or storage space, to more complex platform services like application and other execution environments, to entire software solutions, for example, Web-based office tools. Their growing number comes along with a growing demand for a common way to coordinate and manage interactions of all these services. As the majority of them can be accessed by standard Web-service technology, this demand is basically the demand for orchestrating Web services, a problem for which exactly the field of *business process management* offers solutions. For instance, infrastructures for the *Web Services Business Process Execution Language* (WS-BPEL, short BPEL) provide a platform for executing business processes, or more general, workflows that are based on one or more Web services and that are offered as Web services themselves.

Most of the generic or domain-tailored solutions for creating, executing and managing such *Web-service–based workflows* (e. g., Visual Process Manager[1] and runMyProcess[2]) exhibit sophisticated interfaces, a multitude of connectors to subsystems, and increasing support for non-functional properties such as scalability and security. Nevertheless, fault tolerance has received relatively limited attention. Standard *BPEL engines*, responsible for executing Web-service–based workflows within BPEL infrastructures, log state changes to persistent storage to enable recovery of active workflows after a reboot or crash. This approach has two disadvantages: first, the need for synchronous logging slows down the execution speed during normal operation; second, the reliability of this mechanism depends on the reliability of the storage. In addition, BPEL provides only limited means to handle failures of the Web services the workflows are based on. Making these Web services fault tolerant is not supported at all by standard BPEL infrastructures. However, recent studies on cloud offerings [11] and hardware in general [10] show, that clouds are less reliable than traditional data centers and hardware failures are more common than previously assumed. This basically inhibits outsourcing of critical processes like financial or medical services to clouds.

An effort to close this gap by supporting critical applications in the context of cloud computing is the EU IP project TClouds which targets the provision of a dependable and secure cloud infrastructure. As part of this project, we present a solution for offering highly available, fault-tolerant execution of critical Web-service–based workflows as a platform service within clouds. Our approach is practice-oriented, since current BPEL infrastructures and workflows can be widely reused, and it is extensively configurable as well as

[*] The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°257243 (TClouds project: http://www.tclouds-project.eu/).

CloudCP 2012: 2nd International Workshop on Cloud Computing Platforms. Bern, Switzerland.
Copyright 2012 ACM 978-1-4503-1161-8 ...$10.00.

[1] http://www.salesforce.com/platform/process/

[2] http://www.runmyprocess.com/

dynamically adaptable through the use of external coordination services provided by today's cloud infrastructures.

Unlike other works in this field, the architecture presented here provides fault tolerance by means of active replication at both the process level and the service level. Usually, porting an unreplicated service to run in an actively replicated environment is not trivial and often requires manual modifications to the code, for example, to integrate key mechanisms like message distribution, request ordering, and replica coordination. In contrast, our solution includes an automated and transparent transformation process for workflows that obviates the need for manual intervention. Basically, the transformation process inserts calls to proxy components whenever a Web service is to be invoked, which enables the proxies to intercept calls and carry out tasks necessary for replication. Besides eliminating manual adaption of workflows, this approach permits to reuse standard BPEL engines, simplifying the implementation of our solution. To simplify it further, we heavily make use of Apache ZooKeeper [6], a service to coordinate distributed applications. This allows us to externalize coordination tasks and to realize global configuration as well as dynamic adaptation to changing environmental conditions.

Initial evaluation results show that our approach outperforms a standard BPEL engine, which relies on logging for recovery purposes, by a factor of 2.0 to 3.4 while providing higher availability. Furthermore, the overhead of externalizing coordination is about 13% compared to a traditional design that utilizes a group communication as an integrated part, co-located with the engines.

2. BASICS ABOUT BPEL

The *Web Services Business Process Execution Language* (WS-BPEL, short BPEL) is an XML-based language designed to describe and specify the behavior of *business processes*. In the context of BPEL, a business process is a set of activities that uses different Web services (possibly from different providers) to realize a new *composite Web service*.

Figure 1 shows a standard (unreplicated) BPEL infrastructure. Its main component is the *BPEL engine*, which is responsible for executing business processes specified in *BPEL process definitions*. When a client sends a request to a BPEL service, the BPEL engine handles the request according to the corresponding process definition. It particularly calls all Web services necessary to fulfill the request.

Besides comprising the means to invoke Web services, BPEL also provides mechanisms for holding intermediate data, handling exceptions, and expressing control flows that allow BPEL to be used to describe more complex business processes. Furthermore, most BPEL implementations rely on a recovery mechanism that logs intermediate state on a persistent storage to provide reliable process execution.

Thus, being designed for the description of business processes, BPEL is not confined to this special purpose. It can easily be used to describe all kinds of *workflows*, given that these workflows are offered as and composed of Web services.

3. APACHE ZOOKEEPER

Apache ZooKeeper [6] is a crash-tolerant service, which offers basic services like distributed coordination and configuration maintenance to distributed applications and, among other things, enables the implementation of leader election.

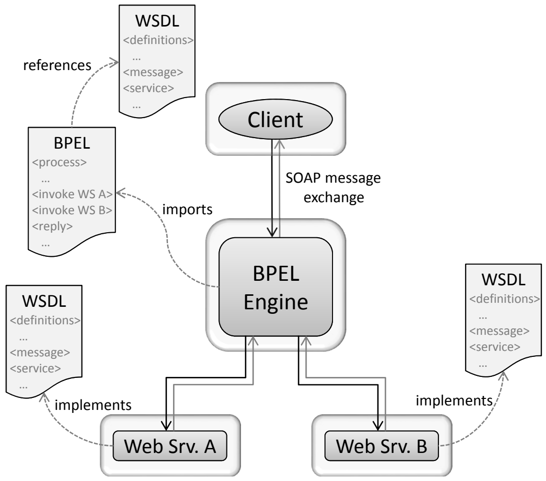


Figure 1: Standard unreplicated BPEL infrastructure: a BPEL engine executes workflows according to their process definitions described in BPEL. In particular, it calls all necessary Web services the workflows are composed of.

ZooKeeper manages data in a hierarchical tree, similar to a file system; each node is accessed using a unique path. All nodes in the tree can store a small chunk of data (usually a few kilobytes). ZooKeeper distinguishes between different types of nodes: *regular nodes* have to be explicitly created and deleted by a ZooKeeper user, *ephemeral nodes* behave like regular nodes except that they are also implicitly deleted when the connection to the user that created the node earlier is terminated. Furthermore, ZooKeeper users are able to request a regular or ephemeral node to be a *sequential node*, which leads to the assignment of a sequence number to the node's name at creation time. In our reliable BPEL infrastructure, for example, we use sequential nodes to establish a total order on the requests of BPEL clients.

ZooKeeper offers a callback mechanism that informs users about certain events (e. g., the creation or deletion of a data node) by registering *watches*. Watches can be used, for instance, to implement fault detection: after establishing a connection, each ZooKeeper user (e. g., a BPEL engine replica) creates an ephemeral node and registers a set of watches to monitor the deletion of the ephemeral nodes of other users. This way, when a user crashes, its ephemeral node is deleted, which in turn triggers the watches that notify all other users about the crash.

Although our current prototype relies on ZooKeeper, the proposed approach is not limited to a specific coordination service implementation. As a consequence, other systems like Chubby [1] could be used for externalizing coordination and configuration in our infrastructure.

4. RELIABLE BPEL INFRASTRUCTURE

Standard BPEL infrastructures have some deficiencies regarding their ability to tolerate faults. Logging intermediate state for recovery purposes, for example, depends on reliable storage. This is a weak point, not only in terms of fault tolerance, but also with regard to performance. Moreover, BPEL processes depend on the Web services they use, but fault tolerance of Web services is not considered at all by standard BPEL infrastructures.

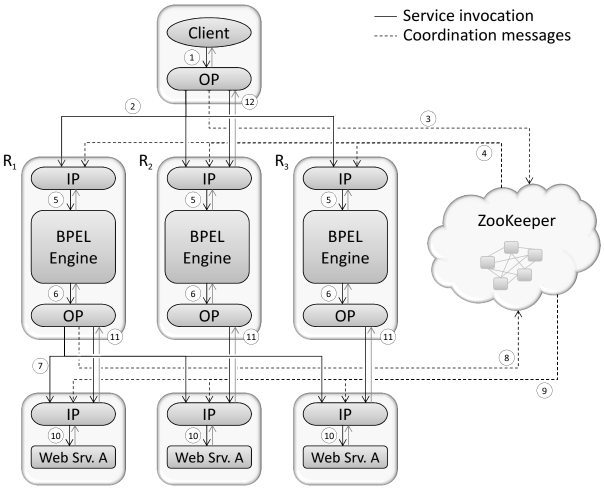


Figure 2: Proposed architecture: BPEL engine and Web services are actively replicated using output and input proxies (OP and IP). An automatic transformation of process definitions allows them to intercept the communication chain transparently. The proxies make use of a ZooKeeper service for coordination and dynamic configuration.

4.1 General Approach

We address these problems by actively replicating not only the BPEL engines, but (optionally) also the Web services in a combined architecture. This architecture (see Figure 2) is designed according to three main objectives. First, all measures taken for fault tolerance have to be transparent to the workflows described in BPEL; that is, it should not make a difference whether a process definition is executed on a standard or a replicated infrastructure. Second, all measures are to be as little invasive as possible to existing BPEL infrastructures in order to reuse them. Third, for further minimizing the implementation effort, cloud services are to be used where possible.

In our architecture, these design objectives are met by means of proxies that intercept Web-service calls to implement replication. In particular, the proxies distribute requests across replicas and collect results. Because Web-service formats and protocols are used between clients and BPEL engines as well as between BPEL engines and Web services, the replication of BPEL engines and Web services can be achieved by almost the same mechanisms.

The proxies use an external ZooKeeper service for coordination, dynamic retrieval of system information and configuration, crash detection, and request ordering. Using an external service simplifies all these tasks and saves resources. It also can permit global coordination within clouds such as the dynamic assignment of replica virtual machines to servers and the consideration of quality-of-service constraints when choosing suitable Web-service implementations.

To integrate a BPEL service with our architecture, the service needs to be made replication aware. This step is performed by a fully automated tool and does not require any manual modifications to the service code. In order to insert calls to the proxies handling replication, our tool transforms the BPEL process definition of the service to integrate. The transformed process definition is then loaded by the BPEL engine replicas. Note that this transformation needs to be executed only once for every service, prior to its first run.

4.2 Replication Architecture

In the following, we describe the steps necessary to process a client request in our reliable BPEL infrastructure.

Client/BPEL Stage. When a client sends a request using a Web-service library, the request is passed to a local *output proxy* (①, see Figure 2). The proxy is responsible for assigning a unique *request id* to the request, sending the request data to all BPEL engine replicas ②, registering the request at ZooKeeper ③, and for collecting the result ⑫. ZooKeeper is used at this stage to obtain the active replicas and to enforce a total order on all requests. The former allows an easy propagation of configuration changes within the system, for example, if replicas fail or are to be replaced. The latter guarantees that all BPEL engine replicas process the same sequence of input data. Sending the request data in a separate step prior to the registration is done for performance reasons, as ZooKeeper is designed for use with small chunks of data. After distributing and registering the request, the output proxy waits until a reply becomes available.

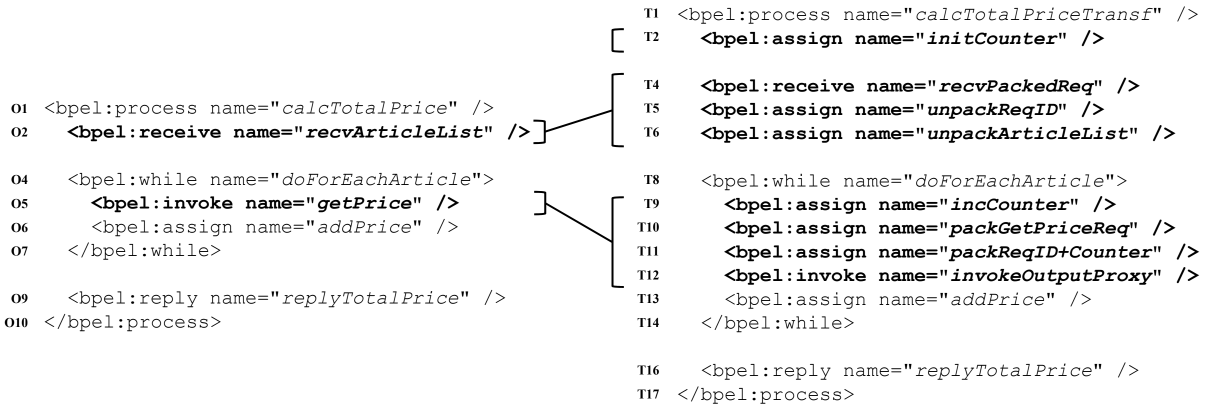
In our architecture, output proxies do not communicate directly with BPEL engines. Instead, each replica runs an *input proxy* in front its BPEL engine. Input proxies maintain a steady connection to ZooKeeper, which allows the detection of crashed replicas. Moreover, they are informed by ZooKeeper when a new request has been registered ④. In that case, they deliver the corresponding request data to the local BPEL engine ⑤ and wait for the reply. After the engine has processed the request, an input proxy stores the reply in a local cache where it can be retrieved by the output proxy of the client.

BPEL/Web-Service Stage. Executing a client request requires a BPEL process to issue calls to different Web services; these calls have to be coordinated across replicas in order to guarantee exactly-once semantics. Similar to the client side, we rely on output proxies for that purpose, to which all outgoing calls of a BPEL process are redirected to ⑥. The redirection of calls is performed statically during the automated process transformation, which also provides the means to tag each Web-service invocation with a *call id* that uniquely identifies each call (see Section 4.3).

The output proxies of all BPEL engine replicas make use of ZooKeeper to elect a leader. The leader (in Figure 2: the output proxy on the left) is responsible for performing the actual calls to Web services, ⑦ to ⑩, analog to the output proxy of a client, ② to ⑤. When a reply becomes available, the output proxies of all BPEL engine replicas retrieve it from a randomly chosen Web-service replica ⑪. Relying on ZooKeeper allows the output proxies to detect the crash of their leader (see Section 3). In such a case, the remaining output proxies elect a new leader which takes over by completing the pending Web-service invocations. With calls being uniquely identifiable, the input proxies of Web services are able to detect and suppress duplicate calls.

4.3 Automated Process Transformation

Actively replicating a service creates the need for mechanisms to distribute messages, order requests, and coordinate replicas. In our architecture, this functionality is provided by means of application-independent proxies (see Section 4.1). In this way, all replication-related measures are completely transparent to the BPEL engines. Thus, existing implementations of such engines can be used in our sys-



```
O1  <bpel:process name="calcTotalPrice" />
O2    <bpel:receive name="recvArticleList" />

O4    <bpel:while name="doForEachArticle">
O5      <bpel:invoke name="getPrice" />
O6      <bpel:assign name="addPrice" />
O7    </bpel:while>

O9    <bpel:reply name="replyTotalPrice" />
O10 </bpel:process>
```

```
T1  <bpel:process name="calcTotalPriceTransf" />
T2    <bpel:assign name="initCounter" />

T4    <bpel:receive name="recvPackedReq" />
T5    <bpel:assign name="unpackReqID" />
T6    <bpel:assign name="unpackArticleList" />

T8    <bpel:while name="doForEachArticle">
T9      <bpel:assign name="incCounter" />
T10     <bpel:assign name="packGetPriceReq" />
T11     <bpel:assign name="packReqID+Counter" />
T12     <bpel:invoke name="invokeOutputProxy" />
T13     <bpel:assign name="addPrice" />
T14   </bpel:while>

T16   <bpel:reply name="replyTotalPrice" />
T17 </bpel:process>
```

Figure 3: Comparison of an original (left) and a transformed (right) process definition (simplified). The business process sums up article prices with the help of a Web service (getPrice) determining these prices. The transformed definition contains steps for unpacking requests and request ids from the input proxy and handling invocations through the output proxy; each call to the external Web service is assigned a unique id.

tem without modifications. This approach, however, makes it necessary to adapt process definition and interface description of each workflow that is to be replicated. Performing this task for every individual workflow by hand is time-consuming and error-prone and therefore not an option. To address this issue, we developed an automated tool that statically transforms all files necessary to make a workflow replication aware prior to its first run.

During the transformation, our tool applies four modifications. First, the interface description of a BPEL process is extended to enable the propagation of request ids. Second, all requests received from clients are unmarshalled to extract the request id attached. Third, all outgoing calls are redirected to an output proxy. Fourth, for each outgoing call to a Web service, the process definition is extended to allow the BPEL process to assign a unique call id to the Web-service call in order to be able to distinguish different invocations (see Section 4.2).

Figure 3 illustrates a simplified example of a transformed process definition for a business process that sums up the prices of different articles; the price of an article is obtained via a `getPrice` Web-service call. In the first step, the transformed BPEL process receives the article list from the client and extracts the request id (L. T4-T6). Next, the process determines the prices of all articles and sums them up (L. T8-T14) before it returns the result to the client (L. T16). Note that the `getPrice` Web-service call in the original process definition (L. O5) is redirected to the output proxy of the transformed BPEL process (L. T12) using a wrapper request that contains the original client request (L. T10) and the call id assigned (L. T11). The call id is dynamically generated by concatenating the request id and the value of a counter which is incremented for every invocation of `getPrice` (L. T9). As a result, the call id allows an output proxy to distinguish different Web-service calls, including multiple calls to the same Web service.

5. EVALUATION

In the course of the evaluation, we pursue two basic questions: what is the overhead for active replication in our approach compared to a standard BPEL infrastructure that uses logging, and what are the costs of externalizing coordination to ZooKeeper? To answer theses questions, we compare our approach with a standard unreplicated BPEL engine and with a variant based on a traditional fault-tolerant architecture using a group communication (i. e., JGroups[3]).

Test Setting. Our test installation comprises 16 machines equipped with 2.4 GHz quad-core CPU, 8 GB RAM, and connected over Gigabit switched Ethernet. As platform for BPEL engines and Web services, we use an Apache software stack containing Tomcat, Axis2 and ODE. The replication groups of BPEL engines, Web services, and ZooKeeper comprise five servers each and are therefore able to tolerate two faults per replica group[4]; the client is executed on a dedicated host. The values presented are the average of multiple test runs, each including 250 requests.

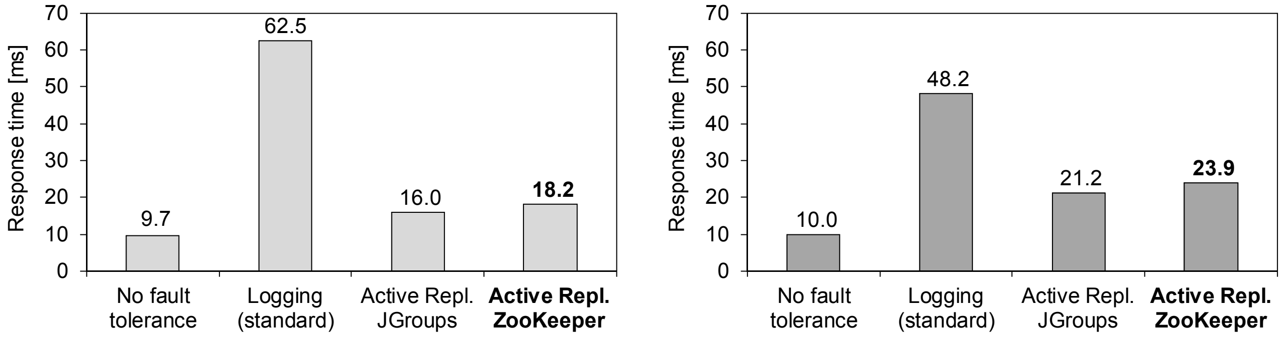
Replication Overhead. As single requests from a client to a BPEL engine typically lead to interaction with multiple Web services, we examine the Client/BPEL stage and BPEL/Web-service stage (see Section 4.1) independently. In the first scenario, we measure the response time to a client request that is immediately answered by a BPEL process, without making any external Web-service calls. In the second scenario, we evaluate the time it takes a BPEL process to receive a reply to a Web-service invocation.

Figure 4 shows that the overhead for replication in our architecture is significantly smaller than the overhead for logging in standard BPEL infrastructures. Without the need to perform costly synchronous logging of intermediate state, our approach achieves 3.4 times (Client/BPEL stage) and 2.0 times (BPEL/Web-service stage) lower response times. The difference between stages is due to the fact that in our implementation the client output proxy is part of the Web-service library whereas the BPEL-replica output proxy is realized as Web service running on the same machine, thus requires an additional Web-service call (see Figure 2, step ⑥).

Disabling the logging mechanism for a standard unreplicated BPEL engine (i. e., deactivating fault tolerance entirely) exposes the high costs of this technique. Without logging state to persistent storage, response times drop to about 10 milliseconds for both stages. However, improving performance this way is not acceptable for critical services, as it prevents a service from being recovered after a crash.

[3] `http://www.jgroups.org/`

[4] Five machines are a common configuration for ZooKeeper.



(a) BPEL echo process called by a client (Client/BPEL stage)

(b) Echo Web service called by BPEL process (BPEL/Web-service stage)

Figure 4: Architecture comparison: unreplicated with and without logging-based fault tolerance vs. transparent active replication via ZooKeeper and JGroups (response times are in milliseconds).

Cost of Externalized Coordination. Instead of externalizing replica coordination to ZooKeeper, we could have used a group communication framework integrated with the input proxies to perform request ordering. To evaluate the effect of our design decision on performance, we implemented a variant of our prototype that is based on the JGroups group communication. In this variant, an output proxy sends a request to only one of the input proxies, not all of them (cf. steps ② and ⑦). The reliable delivery and ordering of requests is subsequently carried out by JGroups. Additional steps (see ③ and ⑧), as needed in the ZooKeeper variant, can be omitted.

A comparison of the results for both variants (see Figure 4) shows that the overhead of using external coordination is moderate (about 13%). Despite the increased number of messages exchanged, the additional latency costs are mostly hidden due to concurrent execution. Even if the underlying network exhibited higher latencies, the overhead should be negligible. The low overhead makes using coordination as an external service attractive, as it simplifies the design and is often already available in cloud infrastructures and data centers, thereby saving on deployment and maintenance costs. Furthermore, management of clouds could take advantage of such an architecture (see Section 7).

Summary. Besides achieving a higher performance than a standard unreplicated BPEL implementation, our fault-tolerant BPEL infrastructure has additional advantages. First, it does not depend on reliable storage. Second, it also provides improved fault tolerance as the services offered by a replicated business process remain available even in the presence of a limited number of crashes.

6. RELATED WORK

Fault tolerance of Web services has been subject of research almost since their first specification. For instance, Dialani et al. [3] designed a framework that implements recovery mechanisms for Web services primarily based on checkpointing and logging at the level of SOAP. Liang et al. [9] added similar mechanisms to the SOAP layer, but used it for passive replication of Web services. Furthermore, they realized different components of their system as Web services themselves, instead of integrating them into existing software stacks as it was done by Dialani et al.

A different approach of how fault-tolerance mechanisms for Web services could be implemented was pursued by Dobson [4]. He analyzed in which way known fault-tolerance patterns can be mapped to and realized with BPEL. However, improving the dependability of BPEL engines was none of his objectives. Lau et al. [8] used BPEL for the active and passive replication of Web services. In order to tolerate BPEL engines failures, they proposed a passive replication scheme. In their concept, a monitoring process at the client is responsible for coordination between engine replicas. This makes the deployment more complex and requires a BPEL engine running at the client, unnecessarily raising the resource demand. Moreover, only stateless Web services are supported and transparent transformation of BPEL process definitions was not addressed by Lau et al., since the system is meant for the fault tolerance of single Web services, not for the fault tolerance of entire Web-service–based workflows.

Fault tolerance of workflows was investigated by Juhnke et al. [7] who implemented a module that is able to transparently handle some faults caused by network or server outages. Here, all Web-service invocations initiated through a BPEL process are intercepted by an extension installed to the BPEL engine. If a failure occurs during an invocation, it is handled by this extension according to policies that take the specific characteristics of a cloud environment into account; the policies are stored in a central registry, which creates a single point of failure. Additionally, the registry is a specialized component, which entails higher development and maintenance efforts compared to the usage of an existing coordination service, while being less flexible. Furthermore, intercepting Web-service calls through extensions binds the solution to a specific BPEL engine. Improving the dependability of BPEL engines was not considered by Juhnke et al.

Di Penta et al. [2] used a different technique for the interception of Web-service invocations. In their architecture, process definitions are transformed to reroute calls to proxies. This is similar to our approach and enables solutions independent of particular BPEL engines. However, their main interest was to establish dynamic binding between workflows and Web services according to functional and non-functional requirements, which can be achieved by solely modifying address information within process definitions. Our architecture, instead, supports comparable mechanisms and additionally permits replication of Web services as well as BPEL engines, requiring the transformation of whole workflows. Moreover, Di Penta et al. create a specialized proxy for each Web service whereas we use generic proxies leading to a simpler deployment and less resource demand.

The concept of generic proxies was applied by Ezenwoye and Sadjadi [5]. Except from that, their solution seems to resemble the solution proposed by Di Penta et al., which particularly means that they did not regard dependability of BPEL engines. They used the proxies to ensure quality-of-service properties and to dynamically adapt BPEL processes to changed environmental conditions.

To our knowledge, the solution presented in this paper is the first implementation of an actively replicated platform for executing Web-service–based workflows with all their components. Realizing active replication transparent to the replicated processes by automatically transforming them is also a novel approach. Furthermore, our solution shows, that the integration of existing coordination services offered by today's cloud infrastructures leads to simplified implementations and new opportunities regarding global coordination and configuration within clouds.

7. FUTURE WORK

The current results for our reliable BPEL infrastructure are encouraging. We therefore intend to extend the presented architecture in several ways:

Using an external coordination service does not only simplify the implementation while causing only moderate overhead, it also creates the possibility of managing entire cloud infrastructures over such a service. Allocation of platforms and resources for virtual machines, failure detection, dynamic reconfiguration, job distribution and control are some exemplary tasks that could be conducted by a cloud-wide coordination service which also acts as a decentralized, partly self-managed registry for global information of all kinds. Although we already utilize this potential, a fair amount of it is still unused, so we plan to exploit it further.

Moreover, distributing all kinds of services over multiple, sometimes up to thousands of hosts increases the risk of arbitrary hardware errors. This is particularly true in the context of cloud computing, where, in general, off-the-shelf hardware is used. Additionally, tolerating only crash failures does not protect against malicious attacks. For these reasons, we will adapt the presented infrastructure in order to tolerate arbitrary, also called Byzantine faults.

An other approach to make the architecture even more dependable is the usage of multiple clouds. Distributing the replicas over different clouds avoids being dependent from a single cloud provider. Thus, we plan to evaluate the behavior of our solution in such an environment.

8. CONCLUSION

BPEL infrastructures provide the execution of business processes that are composed of and offered as Web services. Thus, they would be a perfect supply for the growing demand of Web-service orchestration in cloud computing. However, current BPEL implementations do not support the level of fault tolerance required in this context.

We therefore presented a solution for the provision of platform services within clouds that allows the fault-tolerant execution of critical Web-service–based workflows. In our holistic approach, fault tolerance is achieved by actively replicating not only such workflows, but also the Web services on which they are based and dependent. Using active replication provides very high availability rates, independence of, for instance, reliable storage as mostly required by passive variants, and gives the opportunity to tolerate arbitrary faults in a next step. The use of generic proxies in combination with an automatic transformation of process definitions leads to a simple and practical solution. Existing workflows described in BPEL can be replicated without manual intervention, new workflows can be written as they were intended for an unreplicated operation and current BPEL infrastructures can be reused. The implementation is further simplified by utilizing coordination services offered by today's cloud infrastructures. Externalizing coordination and configuration tasks by means of such services also permits a solution that is dynamically configurable and adaptable according to cloud-wide constraints.

As our evaluation results show, this approach of externalizing coordination incurs only moderate overhead compared to a traditional fault-tolerant architecture, in which a group communication is directly integrated into the middleware, and it outperforms an unreplicated BPEL execution supporting only simple recovery mechanisms.

9. REFERENCES

[1] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*, pages 335–350, 2006.

[2] M. Di Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo, and E. Di Nitto. WS Binder: a framework to enable dynamic binding of composite Web services. In *Proc. of the 2006 Intl. Workshop on Service Oriented Software Engineering*, pages 74–80, 2006.

[3] V. Dialani, S. Miles, L. Moreau, D. D. Roure, and M. Luck. Transparent fault tolerance for Web services based architectures. In *Proc. of the 8th Intl. Euro-Par Conf. on Parallel Processing*, pages 889–898, 2002.

[4] G. Dobson. Using WS-BPEL to implement software fault tolerance for Web services. In *Proc. of the 32nd EUROMICRO Conf. on Software Engineering and Advanced Applications*, pages 126–133, 2006.

[5] O. Ezenwoye and S. M. Sadjadi. TRAP/BPEL: A framework for dynamic adaptation of composite services. In *Proc. of the 3rd Intl. Conf. on Web Information Systems and Technologies*, 2007.

[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. of the 2010 USENIX Annual Technical Conf.*, pages 145–158, 2010.

[7] E. Juhnke, T. Dörnemann, and B. Freisleben. Fault-tolerant BPEL workflow execution via cloud-aware recovery policies. In *Proc. of the 35th EUROMICRO Conf. on Software Engineering and Advanced Applications*, pages 31–38, 2009.

[8] J. Lau, L. C. Lung, J. d. S. Fraga, and G. S. Veronese. Designing fault tolerant Web services using BPEL. In *Proc. of the 7th IEEE/ACIS Intl. Conf. on Computer and Information Science*, pages 618–623, 2008.

[9] D. Liang, C.-L. Fang, C. Chen, and F. Lin. Fault tolerant Web service. In *Proc. of the 10th Asia-Pacific Software Engineering Conf.*, pages 310–319, 2003.

[10] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proc. of the 6th ACM EuroSys Conf.*, pages 343–356, 2011.

[11] K. Sripanidkulchai, S. Sahu, Y. Ruan, A. Shaikh, and C. Dorai. Are clouds ready for large distributed applications? *ACM SIGOPS Operating Systems Review*, 44:18–23, 2010.