Self-Healing Execution of Business Processes Based on a Peer-to-Peer Service Architecture

Thomas Friese[1,2], Jörg P. Müller[2], Bernd Freisleben[1]

[1]Dept. of Mathematics and Computer Science
University of Marburg
Hans-Meerwein-Str, D-35032 Marburg, Germany
{friese,freisleb}@informatik.uni-marburg.de
[2]Siemens AG Corporate Technology,
Intelligent Autonomous Systems,
Otto-Hahn-Ring 6, D-81739 München, Germany
joerg.p.mueller@siemens.com

Abstract. The automated execution of business processes that are composed of individual web services has seen a growing importance throughout enterprise computing in the recent years. The Business Process Execution Language for Web Services (BPEL4WS) has become the predominant language to express such business process compositions. In this paper we present the design and implementation of a Robust Execution Layer that acts as a transparent, configurable add-on to any BPEL4WS execution engine to support self-healing execution of business processes. Resilience of the process execution is achieved through service replacement in case of communication failures, by relying on a robust peer-to-peer service discovery and selection mechanism for alternative services.

1 Introduction

The availability of web-service based middleware has opened new possibilities for business process automation. Web service infrastructures and in particular the WSDL [1] and UDDI [2] standards provide a unified way of describing, registering, and looking up services, and of binding service descriptions to service instances. The service-oriented computing metaphor can be applied in a natural way to model business processes as compositions of individual service requests, which can be mapped to web service calls. The Business Process Execution Language for Web Services (BPEL4WS; in the remainder of this paper abbreviated as BPEL for the sake of simplicity) [3] is probably the best known example of an executable business process language.

A shortcoming of today's business process languages is that the runtime infrastructure supporting them does not provide a great deal of flexibility as to how exceptions and errors are handled. Rather, a lot of the underlying logic in how to deal with failures at execution time needs to be defined at design time and programmed into the process description code.

Over the past few years, peer-to-peer (P2P) computing has been emerging as an architectural approach for building distributed software systems (mostly focusing on distributed resource management) that provides built-in, low-cost, and highly scalable mechanisms for ensuring software resilience.

The objective of the research described in this paper is to bring together the strengths of state-of-the-art service-based business process execution languages and infrastructures (exemplified by BPEL) on the one hand, and of P2P architectures on the other. In particular, we present the design and implementation of a middleware framework called Robust Execution Layer (REL) that acts as a transparent, configurable add-on to any BPEL execution engine to support the self-healing execution of business processes that are managed by the engine. By using P2P protocols managing service registration and lookup, REL provides improved service-level resilience without the explicit need of additional dedicated hardware or communication redundancy, reducing the management overhead for centralized components.

In designing the robust execution layer, a number of technical problems needed to be solved, including the management of the execution context of multiple process instances. The paper presents architectural and methodic approaches to solve these problems. The underlying P2P architecture has been developed in the context of the European Integrated Project ATHENA [4]. ATHENA addresses the vision of seamless interoperation of distributed enterprises across and beyond Europe, focusing on the problem of interoperability, but also covering aspects such as cross-enterprise business process modeling and architectures and platforms for business process management and enactment (see also [5]).

The structure of the paper is as follows: In Section 2, we briefly introduce the BPEL language, discuss levels of resilience, and identify basic requirements and problems to be solved in adding resilience to business process execution. Section 3 introduces the REL architecture and outlines the basic components and their interaction. Section 4 presents an example scenario for the usage of REL. Related work is discussed in Section 5. Section 6 concludes the paper and outlines areas of future work.

2 Problem Description

In this section we will give an introduction to the basic principles of BPEL and analyze a number of problems that need to be addressed to provide self-healing execution support in the case of a partner service failure. Throughout the discussion, we will consider RPC style interactions with partner web services using SOAP [6] encoded messages that are transferred via the HTTP protocol. However the principles discussed in this section can also be applied to message or document based interaction with web services.

2.1 BPEL Basics

The Business Process Execution Language for Web Services has emerged from the earlier proposed XLANG [7] and Web Service Flow Language (WSFL) [8]. It enables the construction of complex web services composed from other web services that act

as the basic activities in the process model of the newly constructed service. BPEL offers a conceptual distinction between *abstract* processes that describe the external view on the process model and *executable* processes that describe the workflow of the compound service and can be executed by a process execution engine in order to provide the functionality of the compound service to a client. The specification of an executable process basically defines a blueprint that models the stateful interaction and is used by the execution engine to derive a *process instance*. This process instance captures the state of the interaction with all external web services and clients as well as internal state data used throughout the process workflow. Access to the process is exposed by the execution engine through a web service interface, allowing those processes to be accessed by web service clients or to act as basic activities in other process specifications.

In traditional workflow management systems, a business process is represented by a workflow model. This model consists of a number of basic activities and describes their order of execution. Similarly, BPEL models business processes as sequences of basic activities and introduces control constructs such as loops or conditional branches [9]. The most important activities offered by BPEL for the business process specification are the *invoke* and *receive* activities. The invoke activity is used to invoke external services while the receive activity enables the process to collect external input and delay further execution of the process flow until reception of this input. In the web service interface exposed by the execution engine, the receive activity is represented by an operation provided to clients to invoke and pass parameter values to the process instance during its execution.

The state of a business process includes the previously exchanged messages between partners as well as temporary data used in the process flow. To catch this state data, BPEL offers the ability to define and modify *variables* in the workflow of a business process. Variables may be typed as WSDL message types, XML Schema [10] simple types or elements.

A number of different process instances derived from the same process specification may be created by the process execution engine upon service requests received from different clients. Messages from the clients to the business process are directed towards a single web service port. While this addressing is sufficient to determine the process specification corresponding to the port or port type, another mechanism is required to identify the correct process instance that should receive the message. BPEL defines the concept of *correlation sets* in order to enable the engine to carry out instance-level routing of messages.

A correlation set is a group of message properties that are sufficient to identify the process instance a message has to be delivered to during the process conversation. The correlation properties can be regarded as late bound constants that are initiated and assigned by a specially marked message.

Activities in a BPEL process are associated with a surrounding scope that holds definitions for variables and correlation sets as well as *event handlers*, *fault handlers* and a *compensation handler*. Event handlers and fault handlers provide a mechanism to respond to messages or faults emitted by activities or external partner services. They are active process logic embedded in the process specification that allows for the termination of activities and the reversal of effects caused by prior execution of activities.

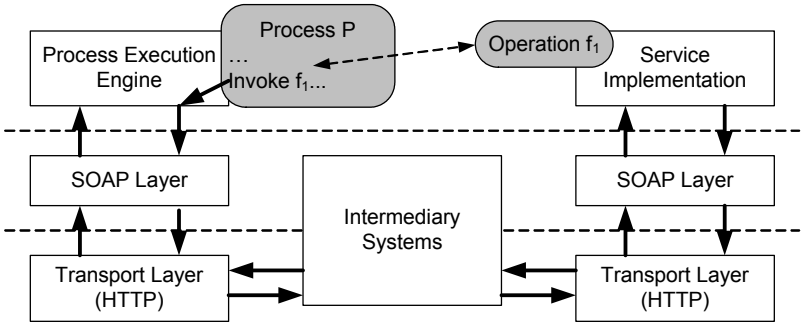

Fig. 1 Process execution environment: The process execution engine encodes service invocations as SOAP messages that are transported via HTTP, received at the service provider after passing possible intermediary systems, decoded and passed to the actual service

2.2 Fault Conditions

Consider the process execution environment shown in Fig. 1.An operation f provided by a partner service is invoked by the process P. The call is encoded as a SOAP message and transmitted via HTTP to the service provider where some service middleware decodes the SOAP message and hands the call to the specific implementation the service. The result of the call is then again encoded as a SOAP reply and transmitted via HTTP to the process engine. Fault conditions can arise at three different levels in this interaction:

- Application specific errors may occur during the processing of the request in the service implementation.
- The service middleware may produce errors during the decoding or encoding of messages if, for example, no suitable serializers are available to encode certain result contents.
- There may be communication failures in the transport protocol. The service provider may not be reachable due to network interruption or system failure.

While errors on the upper two levels are content or application related, a possible recovery from communication failures may be the replacement of the original service by another service that also provides the operation f_1. In this case, the whole process could be successfully finished in spite of communication failures with the originally contacted service provider.

2.3 Realizing of Fault Recovery in the Process Execution Environment

We now present three different ways to realize fault recovery by replacing a service with another service of the same type within the BPEL process execution environment.

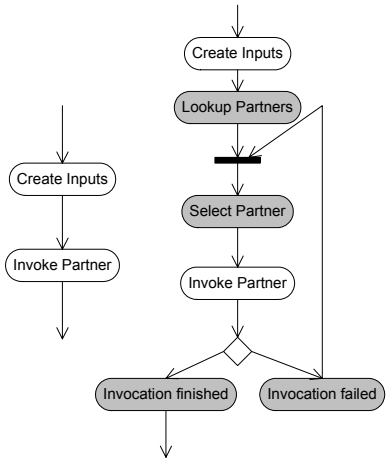

Fig. 2 Structural changes to a process specification when dynamic partner binding is added to an invoke activity.

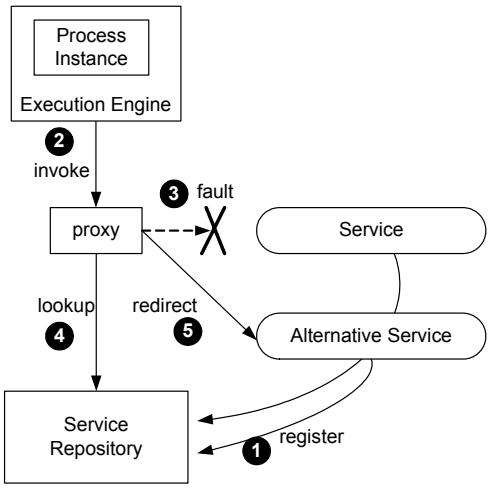

Fig. 3 Redirection of an invocation message after a communications failure and service lookup.

The replacement of a service can be seen as a dynamic partner binding that can be addressed at the process specification level and thus can be encoded in the process specification itself. In this case, the activity to invoke $A.f_1$ is preceded by a directory lookup for services that implement the required port type (i.e. the port type of service A). Then, a loop is added to surround the invoke activity that consecutively replaces the invocation target with one of the services returned by the registry lookup operation until the invoke activity can successfully be carried out. The structural changes to the process specification are shown in Fig. 2. Such a change to the specification of invoke activities would have to be explicitly encoded around every service invocation that is intended to be performed in a more robust way throughout the whole process. We will refer to this alteration of the process specification as *intra- process recovery*.

Similar actions may be implemented in the underlying process middleware (i.e. the process execution engine or the communications infrastructure used to actually transmit the SOAP calls emitted by the execution engine). Communication failures can be detected through expiry of a timeout period. If this event occurs, either (a) the engine or (b) a component in the communication infrastructure may perform registry lookup for alternative invocation targets and reroute the message to an alternative service implementation (this behavior is depicted in Fig. 3). The intermediary system in the communications infrastructure may be implemented as a HTTP or SOAP aware proxy. Therefore, we will refer to the approaches (a) and (b) as *intra-engine recovery* and *proxy recovery*, respectively.

The alteration of the process offers the best way to control the semantics of the compensation action to be associated with an invoke activity. Control of the partner binding can be specified in a very flexible way and the decision to armor certain invocations can be determined on a very fine-grained level. On the other hand, it also poses the need to alter the process specification and introduce additional code at a very fine-grained level. A number of operations need to be introduced for every in-

voke activity leading to process "code" that contains robustness additions eventually exceeding the size of the original process code.

The realization of the recovery strategy within the infrastructure – either as an intra-engine solution or as a proxy solution – does not require changes to the process specification, therefore no additional code has to be introduced in the process specification. The robust invocation of partner services is an inherent, possibly configurable feature carried out in an autonomous manner by the execution environment. In the case of an engine neutral implementation as a proxy instead of an altered specific engine, one is not bound to the concrete implementation or engine features. Furthermore, the feature might be added to the process execution environment without control of the implementation of the process execution engine.

2.4 Dependencies between Activities and Process Instances

In the preceding sections we considered only one invoke activity in the flow of the process. We will now look at an extended example process P_1 that specifies the invocation of a sequence of operations ($A.f_1$, $B.g_1$, $A.f_2$) provided by two partner services A and B. If communication with partner A cannot be established and the infrastructure replaces service A by an alternative service A^* for invocation of the operation f_1, it is very likely that there is an implicit connection between $A.f_1$ and $A.f_2$ that requires the subsequent invocation of f_2 to also be directed towards A^* instead of A. If, for example, A is a hotel accommodation service, f_1 represents a booking function of this service and f_2 a payment verification function, both of these operations have to be used within the same instance of the accommodation service.

The recovery mechanism for invocation failures is built on the assumption that there are at least two distinct instances of the service of type A to choose from. BPEL only offers the options of a static binding to a partner or an explicitly expressed dynamic partner binding through additional mechanisms encoded in the process flow. A conservative and safe assumption for an infrastructure solution that provides invocation robustness through replacement of service instances is to assume that the first invocation of an operation on one service instance selects this service instance for every subsequent invocation. While the process specification determines the type of service to be invoked, the infrastructure holds the ability to select among a set of service instances that implement this service type. After selecting a particular instance, this instance has to be used for every interaction occurring throughout the lifecycle of a process instance derived from the process specification.

It might be desirable to explicitly tighten or relax the service instance binding through internal or external annotation of the business process. Situations may occur where subsequent invocations of operations provided by a service are truly independent and can be directed towards different instances of the service. In other cases, a strict binding to the specified service is desired that should under no circumstances be altered to another service instance implementing the same service type. As an example consider a business process to handle the billing and charging of customers. A contract might bind the service requestor to use the credit card service of one particular company. In this case, it is undesirable to replace the service in case of a communication failure, even if other companies provide an equivalent service.

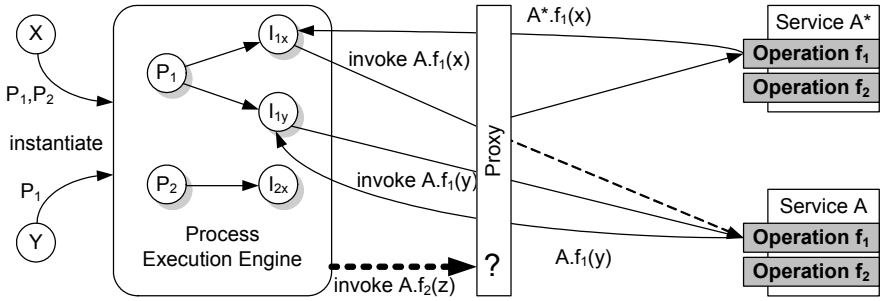

Fig. 4 Two process instances emit a message to operation $A.f_1$. After redirecting one of the invocations, the proxy has to determine an invocation target for a subsequent invocation to $A.f_2$.

Now consider a situation in which two clients X and Y requested the creation of process Instances I_{1x} and I_{1y} derived from P_1. An implementation of the process execution engine has to hold state information for both instances. It can easily extend this state information to incorporate the binding between the process instance and the service instances used. A proxy implementation which is completely decoupled from the process execution engine only receives the SOAP requests emitted by the process execution engine. This information contains the target endpoint used for the communication as well as the message format and message values encoded in the request. Consider the above scenario where I_{1x} and I_{1y} invoke two operations f_1 and f_2 on service A. If the invocation of f_1 gets redirected to A^* for I_{1x} but reaches A in the case of invocation by I_{1y}, the proxy has to identify the process instance that executed the invocation activity in order to determine whether to redirect the call to A^* or directly call A when a subsequent request for $A.f_2$ reaches the proxy. This situation is illustrated in Fig. 4. If another Process P_2 that uses A is instantiated as I_{2x} in the engine, the proxy must also be able to distinguish this instance from the instances derived from the other process specifications.

Information about the process instance may be either explicitly or implicitly encoded in the messages passing through the proxy. An explicit encoding requires the alteration of the business process to emit some instance identification token. This token must be enclosed in every message exchanges with external partner services, therefore the input message format of these services must be altered. This solution is not satisfactory.

A more transparent solution would require the process execution engine to emit the instance identification token transparently to the process and partner services. It could be attached as a SOAP message header to every message emitted by the process execution engine. This mechanism would allow the development of loosely coupled infrastructure components that are not tied to a specific engine implementation but enable these components to distinguish process instances participating in a business interaction. This solution requires a modification of the communication standard implemented by process execution engines.

BPEL addresses a similar problem of instance routing of inbound messages through the concept of correlation sets (see section 2.1). With the definition of *outbound correlation sets* over message properties used in *invoke* activities, a proxy im-

plementation is in a similar way able to correlate a message with the process instance that is the originator of the message exchange. This approach has some limitations: Since BPEL correlation sets are intended to be used on the messages that are inbound to a process, the developer has much greater control over the message specification used in the conversation. For a given set of external services it might be impossible to find a common set of message properties that can be used as a correlation set throughout the whole conversation. Furthermore, a distinction between instances derived from two process specifications that use the same message for initialization of the outbound correlation set is difficult.

3 Design and Implementation of a Robust Execution Layer

In this section, we present the design and implementation of a peer-to-peer based robust execution layer for business processes, addressing the problems and requirements outlined in Section 2. The basic idea of the REL is to provide handling capabilities for low level communication faults in the interaction of a business process engine with external web service providers. By doing so, the business process is protected from failures due to propagated errors that are caused by the low level communication faults.

P2P systems [11] are designed to be self-healing loosely coupled networks of independent nodes. The nodes of the P2P network collaboratively provide a service to each other – such as item storage, lookup and retrieval – that can dynamically adapt to a large number of nodes as well as withstand frequent node arrivals and departures from the network. P2P systems incorporate mechanisms to handle peak loads of information requests as well as sporadic node failures. To discover services, a service repository is needed. In the case of a centralized repository one party has to provide this repository while in the P2P network all involved partners collaborate to operate the needed service repository reducing the maintenance overhead for centralized components in the system. Additionally, P2P schemes have been developed that allow service providers to retain firm control of the information they publish to a service repository. These inherent design principles of P2P systems are ideal properties to be used as an adaptive and resilient information repository in the REL.

The design of the robust execution layer is intended to avoid the need to alter the implementation of the business process execution engine, the business process specification or the implementation of the partner services. As described in Section 2.3, the intra-process as well as the intra-engine recovery realization impose the need for those changes. Therefore, the REL is designed as an intermediary component in the communications infrastructure that intercepts message exchanges between the process execution engine and external services. This intermediary component is basically a SOAP proxy that receives service calls from the process execution engine for added resilience this proxy could in turn be implemented as redundant failover system. As a first step, the proxy has to determine the primary target service for the message. There are two possibilities for the selection of the primary target service:

1. The target service originally requested by the execution engine, if a strict binding to a specific service instance has been specified or no prior interaction between the

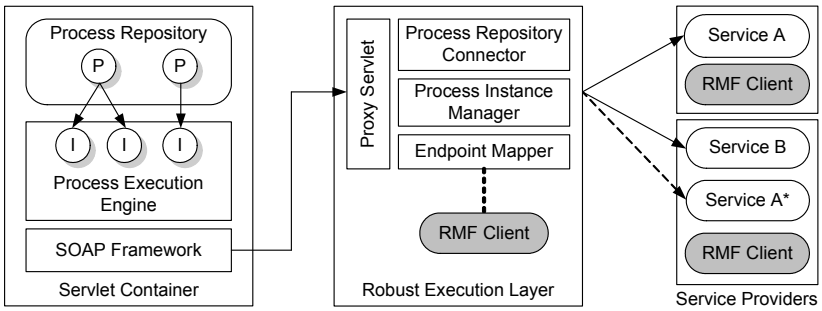

Fig. 5 Components of the Robust Execution Layer acting as an intermediary system between the process execution engine and the service providers.

process execution engine and the service of a specific type has determined a strict binding to another service instance.

2. A different target service that has been selected during prior interaction between the process instance and a specific service instance chosen as an alternative invocation target due to communication faults.

The REL will then try to pass the message on to the primary target service and relay the results of a successful invocation to the process execution engine. Another component of the REL – the *endpoint mapper* - is used to determine a different target service for invocation if no communication link can be established and a deviation from the original invocation target is allowed. This deviation is only allowed in the first case above, when no strict binding has been specified (e.g. through process annotation). The question whether an alternative target of the invocation is required, allowed or prohibited is answered by the *process instance manager* of the REL. The attempt to contact different implementations of a specific service type may be repeated upon subsequent communication faults. If no link can be established at all, the error condition is passed on to the process execution engine where fault handling mechanisms specified for the business process have to ultimately deal with the error.

The purpose of the endpoint mapper is the discovery of alternative services for a given target endpoint. It first needs to determine the type of the target service (i.e. the port types implemented by the service that is associated with the original target endpoint). Afterwards, it must query a service repository to find other services that implement the same port types. A centralized service repository is a single point of failure in the overall system. Therefore, the endpoint mapper of our robust execution layer is implemented on top of a P2P system that provides mechanisms for the storage and retrieval of key-value-pairs in a robust way even under node failure.

For the purpose of instance identification and the initial identification of partner services associated with the business process specifications we use a *process repository connector* to realize access to the set of process specifications that have been deployed in the business process execution engine. Access to the process specifications is optional and only needed to allow an implementation of the *process instance manager* to determine the process instance that is the originator of a specific service call.

The resulting component design of the REL with an embedded business process execution engine as well as some partner service providers is illustrated in Fig. 5. The

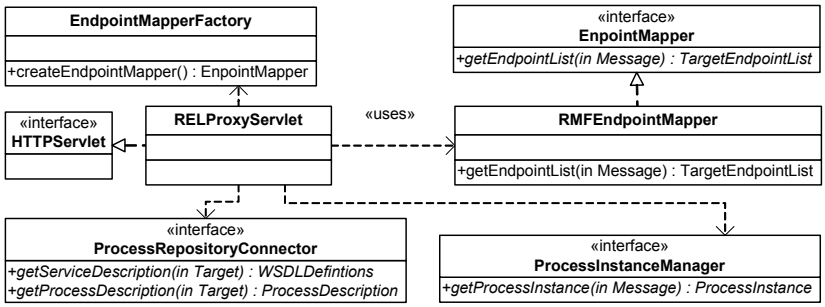

Fig. 6 Core interfaces used by the REL proxy. The factory pattern is used for all components in analogy to the EndpointMapperFactory.

core interfaces of the central components used by the REL proxy are illustrated in Fig. 6. Supporting classes as well as the factories for the process instance manager as well as the process repository connector components are omitted for brevity.

We use the IBM Business Process Execution Language for Web Services Java Run Time (BPWS4J) [12] as the process execution engine with our implementation of the REL. The engine is deployed in a Tomcat servlet container [13] that is configured to use the REL implementation as a proxy for HTTP requests. This is simply achieved by passing the `http.proxyHost` and `http.proxyPort` options to the java virtual machine that is used to load the Tomcat server.

The functionality of the REL is provided to the process execution engine by an implementation of an HTTPServlet. This `RELProxyServlet` works as a transparent HTTP proxy when all robust execution support is disabled. In order to provide the additional functionality, the `RELProxyServlet` uses Factories to instantiate the needed components. The concrete implementation of the corresponding interfaces of these components is configurable by specifying the implementing classes as run time properties. The basic algorithm used by the RELProxyServlet to handle a client request is shown in Fig. 7.

3.1 P2P Based Service Discovery

A robust lookup component for partner services is a key concept of the REL. In a classical service-oriented implementation of the system, this functionality is provided by a centralized service repository. To construct a more robust system, we implemented this service repository using the Resource Management Framework (RMF) [14].

The RMF is a P2P system that collects a number of nodes to logically provide a single distributed hash table called the *Information Space* of the RMF. This information space allows to publish and retrieve data elements that are called *resources*. Leasing and replication are used to ensure persistence of the published information even under conditions of node failure. The RMF provides mechanisms to modify and search for resources in the information space and to subscribe to resource changes on elements already published or published in the future. Resources in the RMF are XML elements that have the following child elements:

- The mandatory ID of a resource that is used to uniquely identify the element. This can either be a globally unique UUID or some fully qualified hierarchical name, guaranteeing uniqueness of the ID.
- An optional name of the resource to be used as a user friendly name for application independent presentation of the resource.
- An optional list of keywords to be associated with the resource.
- Any number of application specific XML elements.

Developers are free to determine the values of the root element as well as its namespace specification.

When a resource is published in the RMF information space, an internal mapping to peer addresses in the system is calculated to get a list of nodes that are ultimately used to store the resource. The ID and keywords associated with a resource are used to calculate this mapping. The search operation works in two phases. It is directed towards a set of peers that is determined by the same mapping calculation based on a set of given keywords or resource IDs. Afterwards, a query is directed towards this set of peers in order to find the desired information among all resources stored at the specified peers. The XPath query language [15] can be used to formulate queries for resources.

```
receive message
pi = processInstanceManager
     .getProcessInstance(message)
if ( pi.mustRedirect(message.target) )
mesage.changeTarget(
          pi.getRedirectionTarget(message))
deliver(message)
if ( ! message.delivered() )
     targetServiceList = endpointMapper
                    .getEndpointList( message )
while ( pi.allowRedirect(message.target)
        AND
        targetServiceList.hasMoreTargets()
        AND
        ! message.delivered() )
{
        message.changeTarget(
            targetServiceList.getNextTarget() )
        deliver(message)
}
```

Fig. 7 Core message handling algorithm used by the REL proxy servlet.

The web service description language is an XML format used to describe web services as a set of endpoints, operations and message formats used in the communication with the service. We have defined a resource format to publish WSDL descriptions of web services in the RMF information space. This mapping from WSDL descriptions to RMF resources is tailored towards answering the query stated by the REL endpoint mapper. A WSDLResource contains the WSDL document as a child element, a generic UUID for identification purposes and the names of the port type, operations and service elements as well as the endpoint address associated with the port definitions of the WSDL document as keywords for the resource.

We use the WSDL4Java [16] API to deserialize WSDL documents into in memory object representations that are then traversed to collect the needed keywords. The REL implementation provides the `WSDL2RMF` class that exposes a set of static `createResource` methods when a WSDL description is either passed as String, Stream or URL reference. The WSDLResource returned by this implementation can then directly be published using the RMF API. The WSDL to RMF mapping is also needed by service providers that wish to publish their services in the information space. Implementing a P2P client that connects to the RMF information space and publishes WSDLResources for a given set of WSDL descriptions is straightforward and requires only a few lines of Java code.

A RMF based implementation of the EndpointMapper interface is provided by the RMFEndpointMapper class. It uses the search method of the RMF API in order to locate the WSDLResource associated with the endpoint address specified in the service invocation that was received by the REL proxy. After retrieval of the WSDLResource from the information space, the endpoint mapper can determine the port type implemented by the target service and issue a second query to the information space that is now based on the port type name. This second query yields a list of registered services that implement the desired port type and are then returned as a list of alternative invocation targets. If the lookup based on the target endpoint address fails (i.e. the service has been unavailable for a long time and the WSDL resource has been pruned from the information space by leasing), the process repository locator is used to determine the port type of the target service.

3.2 Process Repository Connector

The BPWS4J engine uses a set of java server pages for the deployment and removal of processes. Our implementation of the process repository connector is an extension of this management application. It extends the deployment and undeployment functionality by taking a snapshot of the BPEL process specification to be used by the REL. In addition to the process description, the WSDL descriptions of the partner services involved in the business process are captured by the process repository connector.

The BPEL specification respectively the WSDL descriptions are parsed to collect a set of port type to service endpoint mappings that can later on be used if the port type of a service can not be resolved by a query to the RMF. For this purpose, the repository connector exposes the method `getServiceDescription` that takes a target endpoint specification as input. Our prototypical implementation of the process in-

stance manager uses the process repository connector's functionality to gain knowledge about the process descriptions to enable process instance distinction based on the messages used to invoke partner services.

3.3 Process Instance Manager

The basic functionality of the REL implementation has been tested using a single process instance in the BPWS4J engine. In this limited setting, no explicit distinction of the process instance is needed. A first prototypical implementation of the process instance manager has been created to experiment with the usage of instance tokens emitted as SOAP headers by the process execution engine.

Additionally, a first implementation of a process instance manager has been developed that uses the process specification returned by the process repository connector in conjunction with the definition of outbound correlation sets. In this case, the process instance manager parses the SOAP messages to identify the message properties included in the correlation set definitions.

4 Usage Scenario

Using the REL has a potential impact on the business conversation governed by a business process description both for the initiator and for the external service providers. We will discuss motivations for using the self-healing behavior of the REL for both parties in the setting of a travel planning scenario. We selected this scenario because it is easy to understand, and often referred to in the literature [17][1].

Assume that a number of individual service providers offer web services that enable customers to search for hotel rooms, flight and car rental offers and book them. A travel agency may describe a business process (referred to as travel process) that composes these services into a new service that is capable of offering full travel packages including airline tickets, hotel accommodation and a rental car.

If the hotel booking service fails during the lifetime of an instance of this travel process, no offer for a travel package can be made to the customer, assuming a standard modeling of this process in BPEL. This is true even if another hotel booking service provider offers the same service and a solution would exist in principle. Using the REL this other service provider would be contacted allowing the travel agency to provide the travel offer. This way, we can reduce the risk of process failure and thus enhance customer satisfaction. While the benefit for the user of the services is obvious, the service providers might be reluctant to participate in the system since it might enable their customers to dynamically switch to another provider on system failure. We believe that service providers will nevertheless accept this, since they effectively participate in a marketplace where robustness against service failure can be a competitive feature. In order to make its web services resilient against failure, one provider may install a high availability or load-leveling system. This is basically a pro-

[1] In the ATHENA project, we are applying the REL in a more complex automotive supply chain application.

vider side proxy that uses a number of backend service providers to relay client requests to. Instead of using such an expensive and hard to maintain solution, the service provider may also directly publish the services available at the backend layer into the P2P repository where the REL can find the set of backend servers and use them accordingly. The two approaches are illustrated in Fig. 8. The REL approach has been implemented for this sample use case, and basic functionality could be shown.

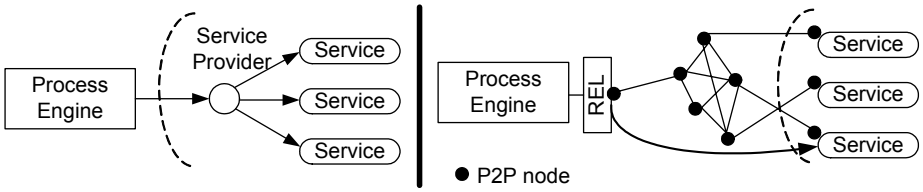

Fig. 8 Service Provider using dedicated load balancing and high availability component on the left side and direct service publishing in the P2P infrastructure on the right side.

5 Related Work

The P2P based lookup mechanism for web services is a key component for the REL. In [18] an approach to web service discovery in P2P indices based on space filling curves is presented that allows for range queries about keywords. Currently only literal keyword queries are needed to support the REL, additionally range queries over keywords are currently being implemented in the RMF – our underlying P2P information infrastructure. Other approaches for decentralized web service discovery focus on the use of ontologies and service semantics to organize web service registries into collections [19] or perform web service discovery on the semantic web [20], this is not the focus of our work.

In [21] a framework for autonomic modeling and simulation of business processes is proposed, this work focuses on supporting the design and development of business processes, not on autonomic process execution.

General requirements for self-healing system architectures are analysed in [22]. Robustness against external or internal failure is one of the relevant requirements to achieve a self-healing architecture style, it is also a feature our REL introduces to the process execution environment. In [23] the authors quantify the effectiveness of self-healing strategies used within service discovery systems, our work focuses on the architectural extension of the business process execution environment to achieve self-healing capabilities. The self-healing service discovery mechanism is only a part of that work that we assume to be addressed in the underlying P2P system. The work in [24] proposes a path to a more autonomic behavior of web services, a general extension of the service oriented archictecture is proposed that does not address the requirements of business process execution.The author of [25] evaluates different message passing strategies for optimization of message flow in process based EAI systems, he does not show an architecture that generally offers self-healing capabilities for different process execution engines.

6 Conclusions

The research described in this paper is motivated by the idea of introducing self-healing mechanisms to business process execution by integrating state-of-the-art service-based business process execution languages and infrastructures (exemplified by BPEL) on the one hand, and of P2P architectures on the other. The main contribution of this paper is twofold: Firstly, we presented an analysis of the shortcomings of existing business process execution frameworks concerning flexible failure handling; secondly, we presented the design and implementation of a middleware framework called Robust Execution Layer that acts as a transparent, configurable add-on to any BPEL execution engine to support the self-healing execution of business processes that are managed by the engine. The combination of BPEL with Siemens' P2P Resource Management Framework enables service-level resilience without the explicit need of additional dedicated hardware or communication redundancy, and transparently supports different underlying software architectures.

We will further investigate the problem of process instance identification at the service level in a loosely coupled infrastructure setting in the future. We believe that P2P computing offers an interesting architectural approach to leverage the functionality of today's client-server business process engines to the case of cross-organizational business processes that are characterized by heterogeneity, constant change, autonomy of partners, and limited information/service access due to organizational boundaries and competition. This paper investigated the application of P2P resource management to the *service* level. Thus, an important aspect for our future research is to extend the scope of this work by investigating the applicability of P2P concepts to other facets of distributed business resource management, including *business objects* (e.g. the secure seamless access to business documents such as a request for quotation in a sourcing application, or a technical specification in a collaborative product design scenario) and *processes* (e.g. bottom-up organization of a cross-organizational business process through the P2P interaction of multiple business process engines). Furthermore, it would be interesting to investigate the potential of applying the REL in service oriented grid computing environments [26] when modeling grid applications as process oriented service compositions.

Acknowledgements

Part of the work reported in this paper is funded by the E.C. within the ATHENA IP under the European grant FP6-IST-507849. The paper does not represent the view of the E.C. nor that of other consortium members, and the authors are responsible for the paper's content.

7 References

1. W3C Recommendation: "WSDL 1.1", http://www.w3.org/TR/wsdl, 2001.
2. "Universal Description, Discovery and Integration", Technical White Paper, 2000.

3. IBM (2003) “BPEL4WS: Business Process Execution Language for Web Services”, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/
4. ATHENA. Advanced Technologies for Interoperability of Heterogeneous Enterprise Networks and their Applications. European IP FP6-IST-507849. http://www.athena-ip.org.
5. Müller, J. P., Bauer, B., Friese, T.: ”Programming Software Agents as Designing Executable Business Processes: A Model-Driven Perspective“. In Proceedings of the 1st Int. Workshop on Programming Multi-Agent Systems, Melbourne, Australia, Vol. 3067 of Lecture Notes in AI, Springer-Verlag, 2004.
6. W3C Recommendation: “SOAP Version 1.2”, http://www.w3.org/TR/SOAP, 2003.
7. Microsoft (2001): “XLANG – Web Services for Business Process Design”.
8. IBM (2001):”Web Services Flow Language”.
9. Khalaf, R., Mukhi, N., Weerawarana, S.: “Service-Oriented Composition in BPEL4WS”. In: Proceedings of The Twelfth International World Wide Web Conference, 2003.
10. W3C Recommendation, “XML Schema”, 2001.
11. Oram, A.: “P2P: Harnessing the Power of Disruptive Technologies”, O'Reilly, 2001.
12. IBM Business Process Execution Language for Web Services Java Run Time. http://www.alphaworks.ibm.com/tech/bpws4j
13. Apache Software Foundation: “Apache Jakarta Tomcat 5.x”.
14. Friese T., Freisleben B., Rusitschka S., Southall A.: “A Framework for Resource Management in Peer-to-Peer Networks”, In Proceedings of NetObjectDays 2002, Volume 2591 of Lecture Notes in Computer Science, pp. 4—21, Springer-Verlag.
15. W3C Recommendation “XML Path Language (XPath), Version 1.0”, 1999.
16. IBM (2003), ”The Web Services Description Language for Java Toolkit” JSR110 reference implementation, http://www-124.ibm.com/developerworks/projects/wsdl4j/
17. Ingham, D., Caughey, S., Watson, P., Halsey, S.: “The Informed Traveller: A Case Study in Building Internet Brokering Services”, Proceedings of the IEEE Workshop on Internet Applications, 1999, p. 44.
18. Schmidt, C., Parashar, M.: “A Peer-to-Peer Approach to Web Service Discovery”, World Wide Web Journal, Volume 7, Issue 2, June 2004, pp. 211 – 229.
19. Sivashanmugam, K., Verma, K., Mulye, R., Zhong, Z., Sheth, A.: “Speed-R: Semantic P2P Environment for diverse Web Service Registries”.
20. Schlosser, M., Sintek, M., Decker, S., Nejdl, W.: „A Scalable and Ontology-Based P2P Infrastructure for Semantic Web Services”. Second International Conference on Peer-to-Peer Computing (P2P’02), 2002.
21. Yu, X., Zhang, L., Li, Y., Chen, Y.: „WSCE: A Flexible Web Service Composition Environment“, Proceedings of the International Conference on Web Services, San Diego, California, 2004, p. 428.
22. Mikic-Rakic, M., Mehta, N., Medvidovic, N.: “Architectural Style Requirements for Self-Healing Systems”, In Proceedings of the First Workshop on Self-healing Systems, Charleston, South Carolina, 2002, pp. 49 – 54.
23. Dabrowski, C., Mills, K.: “Understanding Self-healing in Service-Discovery Systems”, In Proceedings of the First Workshop on Self-healing Systems, Charleston, South Carolina, 2002, pp. 15-20.
24. Birman, K., Renesse, R. van, Vogels, W.: “Adding High Availability and Autonomic Behavior to Web Services”, In Proceedings of The 26th International Conference on Software Engineering, Edinburgh, Scotland, United Kingdom, 2004, pp. 17 – 26.
25. Caseau, Y.: “Self-Adaptive and Self-Healing Message Passing Strategies for Process-Oriented Integration Infrastructures”, In Proceedings of The 11th Int. Conference on the Engineering of Computer-Based Systems, Brno, Czech Republic, 2004, pp. 506 – 512.
26. Smith, M., Friese, T., Freisleben, B.: “Towards a Service-Oriented Ad Hoc Grid”, Proceedings of the 3rd International Symposium on Parallel and Distributed Computing, Cork, Ireland, 2004.