An Environment for Flexible Advanced Compensations of Web Service Transactions

Schaefer, Michael; Dolog, Peter; Nejdl, Wolfgang

Published in:
ACM Transactions on the Web

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

An Environment for Flexible Advanced Compensations of Web Service Transactions[1]

MICHAEL SCHÄFER
L3S Research Center, University of Hannover
PETER DOLOG
Department of Computer Science, Aalborg University
and
WOLFGANG NEJDL
L3S Research Center, University of Hannover

Business to business integration has recently been performed by employing Web service environments. Moreover, such environments are being provided by major players on the technology markets. Those environments are based on open specifications for transaction coordination. When a failure in such an environment occurs, a compensation can be initiated to recover from the failure. However, current environments have only limited capabilities for compensations, and are usually based on backward recovery. In this paper, we introduce an environment to deal with advanced compensations based on forward recovery principles. We extend the existing Web service transaction coordination architecture and infrastructure in order to support flexible compensation operations. A contract-based approach is being used, which allows the specification of permitted compensations at runtime. We introduce the *abstract service* and *adapter* components which allow us to separate the compensation logic from the coordination logic. In this way, we can easily plug in or plug out different compensation strategies based on a specification language defined on top of basic compensation activities and complex compensation types. Experiments with our approach and environment show that such an approach to compensation is feasible and beneficial. Additionally, we introduce a cost-benefit model to evaluate the proposed environment based on net value analysis. The evaluation shows under which circumstances the environment is economical.

Categories and Subject Descriptors: B.1.3 [**Control Structure Reliability, Testing, and Fault-Tolerance**]: Diagnostics, Error Checking; C.2.4 [**Distributed Systems**]: Distributed Applications; C.4 [**Performance of Systems**]: Fault tolerance, Reliability, availability, and serviceability; H.3.4 [**Systems and Software**]: Distributed systems, Information networks; H.3.5 [**Online Information Services**]: Web-based services

General Terms: Design, Reliability

Additional Key Words and Phrases: Web Services, Transactions, Compensations, Forward-Recovery

[1]A preliminary version of this paper appeared in proceedings of ICWE 2007 [Schäfer et al. 2007]

Authors' addresses: Michael Schäfer and Wolfgang Nejdl, L3S Research Center, University of Hannover, Appelstr. 9a, D-30167 Hannover, Germany, `Michael.K.Schaefer@gmx.de, nejdl@l3s.de`
Peter Dolog, Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark, `dolog@cs.aau.dk`

1. INTRODUCTION

The Web service environment has become the standard for Web applications supporting business to business transactions and user services. Processes such as payroll management or supply chain management are realized through Web services. In order to ensure that the results of the business transactions are consistent and valid, Web service coordination and transaction specifications [Arjuna Technologies Ltd. et al. 2005; Arjuna Technologies Ltd. et al. 2005; Arjuna Technologies Ltd. et al. 2005] have been proposed. They provide the architecture and protocols that are required for transaction coordination of Web services.

The transaction compensation [Gray 1981] is a replacement for an operation that was invoked but failed for some reason. The operation which replaces the original one either undoes the results of the original operation, or provides similar capabilities as the original one. The notion of compensation was introduced for environments where the isolation property of transactions is relaxed but the atomicity needs to be maintained. Several protocols have been proposed to control transactional processes with compensations [Yang and Liu 2006].

Current open specifications for transaction management in Web service environment provide only limited compensation capabilities [Greenfield et al. 2003]. In most cases, the handling of a service failure is restricted to *backward recovery* in order to maintain consistency, i.e. all running services participating in the transaction are aborted, and all already performed operations are reversed [Alonso et al. 2003]. Subsequently, the aborted transaction will usually have to be restarted and all requests resend to the Web services, because the failed distributed application still has to perform its tasks. Backward recovery therefore results in the loss of time and money that has already been spent in the aborted transaction, and additional resources are needed to restart the transaction. Moreover, the provider of the service that has encountered an error might have to pay contractual penalties, because the failure has violated the *Service Level Agreement (SLA)* he has with the client. The rollback of the complete transaction due to the failure of one single participating service can also have widespread consequences: All *dependent transactions* on the participating Web services (i.e. transactions that have started operations on a service after the currently aborting transaction and therefore have a completion dependency [Choi et al. 2005]) have to abort and perform a roll back. Therefore, using the backward recovery approach, the failure of one single participating Web service can trigger the abort of many transactions and thus lead to *cascading compensations*, which can result in a huge loss of time and money. This is sometimes called the *domino effect* [Pullum 2001].

In addition to the problematic consequences of backward recovery, current approaches do not allow any changes in a running transaction. If for example erroneous data was used in a part of a transaction, then the only possible course of action is to cancel the transaction and to restart it with correct data.

In this paper, we describe an environment for advanced compensation operations adopting *forward recovery* within Web service transactions. Forward recovery proactively changes the state and structure of a transaction after a service failure occurred, and thus avoids having to perform a rollback and enables the transaction to finish successfully. The main idea is the introduction of a new component called an *abstract service*, which functions as a mediator for compensations, and thus hides the logic behind the introduced compensations. Moreover, it specifies and manages potential replacements for primary Web services to be used within a transaction. The compensations are performed according to predefined rules, and are subject to contracts [Meyer 1992]. We introduce a framework based on the abstract services, which enables the compensations described in the compensation specifications [Schäfer et al. 2007].

Such a solution has the following advantages:

—Compensation strategies can be defined on both, the service provider and the client side. They utilize local knowledge (e.g. the provider of a service knows best if and how his service can be replaced in case of failure) and preferences, which increases flexibility and efficiency.

—The environment can handle both, internally and externally triggered compensations.

—The client of a service is informed about complex compensation operations, which makes it possible to trigger additional compensations. Compensations can thus consist of multiple operations on different levels, and consistency is achieved through well defined communication protocols.

—By extending the already adopted Web service specification, it is not necessary to discontinue current practices if compensations are not required.

—The separation of the compensation logic from the coordination logic allows for a generic definition of compensation strategies, independent from the coordination specification currently in use. They are therefore more flexible and can easily be reused in a different context.

Furthermore, we evaluate the environment according to a cost benefit model as well as several experiments. The evaluation shows under which circumstances the proposed environment is beneficial.

The remainder of the paper is structured as follows. Section 2 introduces the motivating scenario, which will be used in the paper in order to exemplify the concepts. Section 3 introduces design and prototype implementation of an infrastructure that is able to handle internally and externally triggered compensations without transaction aborts, and describes the basic components and compensation specifications based on compensation activities and compensation types. Section 4 discusses various aspects of the evaluation we have performed with the proposed environment. This includes a case study according to the motivating scenario as well as various experiments showing the performance of the environment. The new design is also evaluated analytically on the basis of a net value model. Section 5 reviews related work in the area of forward recovery. Section 6 concludes this paper and provides a direction for future work on this topic.

2. MOTIVATING SCENARIO

The motivating scenario for this paper is a company's monthly payroll processing. In order to introduce real-life dependencies, both, the company's and the employee's responsibilities are considered.

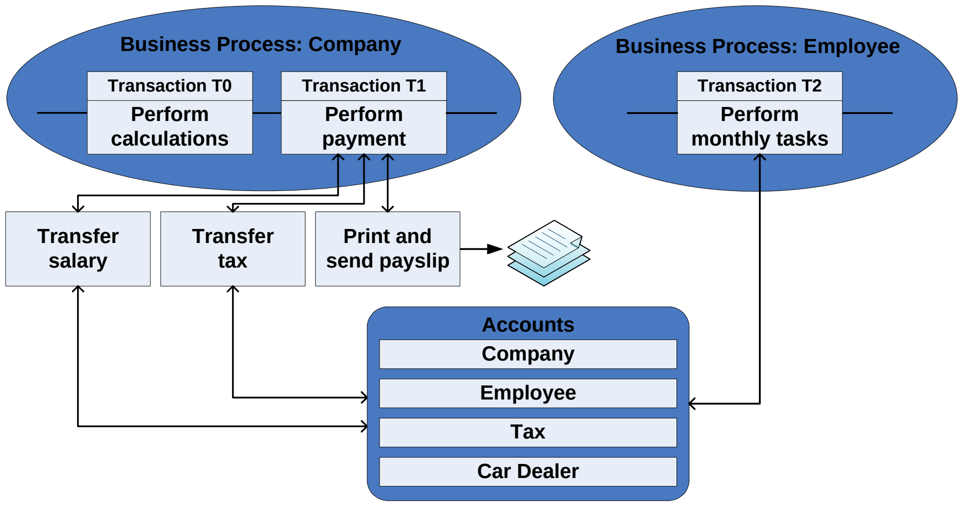

Fig. 1. The motivating scenario

Company: In the first step of the payroll processing procedure, the company has to calculate the salary for each employee, which can depend on a multitude of factors like overtime hours or bonuses. In the next step, the payment of the salary is performed, which comprises several operations. First of all, the salary is transferred from the company's account to the employee's account. Then the company transfers the employee's income tax to the account of the fiscal authorities. Finally, the company prints the payslip and sends it to the employee.

Employee: The employee has only one task which he has to perform each month in this scenario: He transfers the monthly instalment for his new car to the car dealer's account.

The company's and the employee's operations are each controlled by a business process, and are implemented using Web services from multiple providers. The two business processes use transactions in order to guarantee a consistent execution of all required operations. This is depicted in Figure 1. Only the services of transaction T1 are shown.

It is obvious that there are multiple dependencies in this simple scenario, between and within these transactions. Therefore, it is vitally important that no transactions have to be aborted and compensated in order to avoid cascading compensations. However, such a situation can become necessary quite easily:

(1) It can always happen that a service which participates in a transaction fails. Here, it could be that the service that handles the transfer of the salary fails due to an internal error. The transaction inevitably has to be aborted, even though the error might be easily compensatable by using a different service that can perform the same operation. Such a *replacement* is encouraged by the fact that usually multiple services exist that have the same capabilities.

(2) A mistake has been made regarding the input data of an operation. In this scenario, it could be that the calculation of the salary is inaccurate, and too

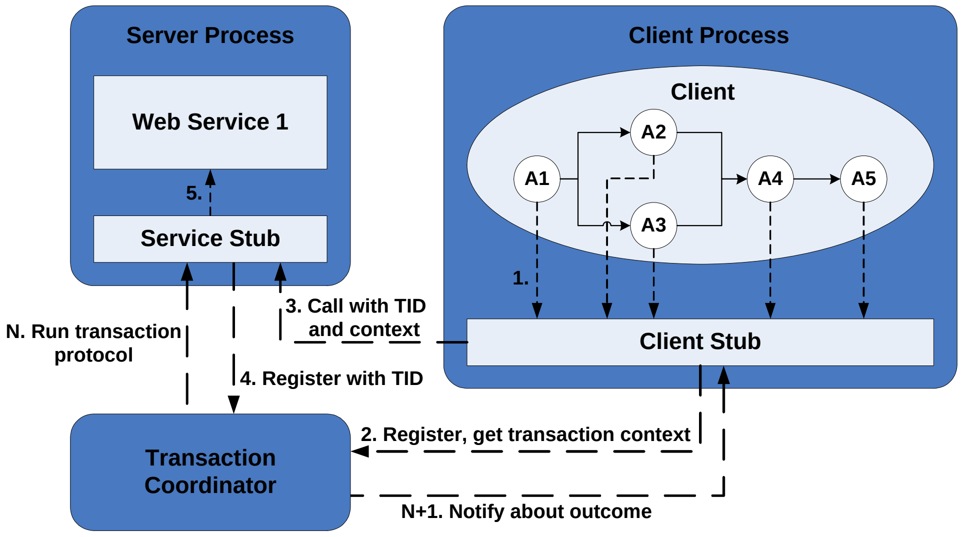

Fig. 2. Transactional environment for Web services adopted from [Alonso et al. 2003]

much has been transferred to the employee's account. The flaw is spotted by an administrator, but the only option is again to abort the complete transaction, although it would be very easy to correct the mistake by transferring the sum that has been paid too much back to the company's account.

Although it should be possible to handle these situations without the need to cancel and compensate the transaction(s), current technology does not allow to do so in a sensible way.

3. WEB SERVICE ENVIRONMENT WITH TRANSACTION COORDINATION

We base our work on Web service coordination and transaction specifications [Arjuna Technologies Ltd. et al. 2005; Arjuna Technologies Ltd. et al. 2005; Arjuna Technologies Ltd. et al. 2005]. These transaction specifications provide a conceptual model and architecture for environments where business activities performed by Web services are embedded in transactional contexts.

Figure 2 depicts an excerpt of such an environment with the main components. The client runs business activities A1 to A5, which are embedded in a transactional context. The transactional context and conversation is maintained by a transaction coordinator. Client and server stubs are responsible for getting and registering the activities and calls for Web services in the right context. The sequence of conversation messages is numbered. For clarity, we only show a conversation with a Web service provider that performs business activity A1. The transaction coordinator is then responsible for running appropriate protocols, such as two phase commit or some of the distributed protocols for Web service environments such as [Alrifai et al. 2006].

As pointed out above, the compensation capabilities are left to the client business activities according to the specifications in [Arjuna Technologies Ltd. et al. 2005;

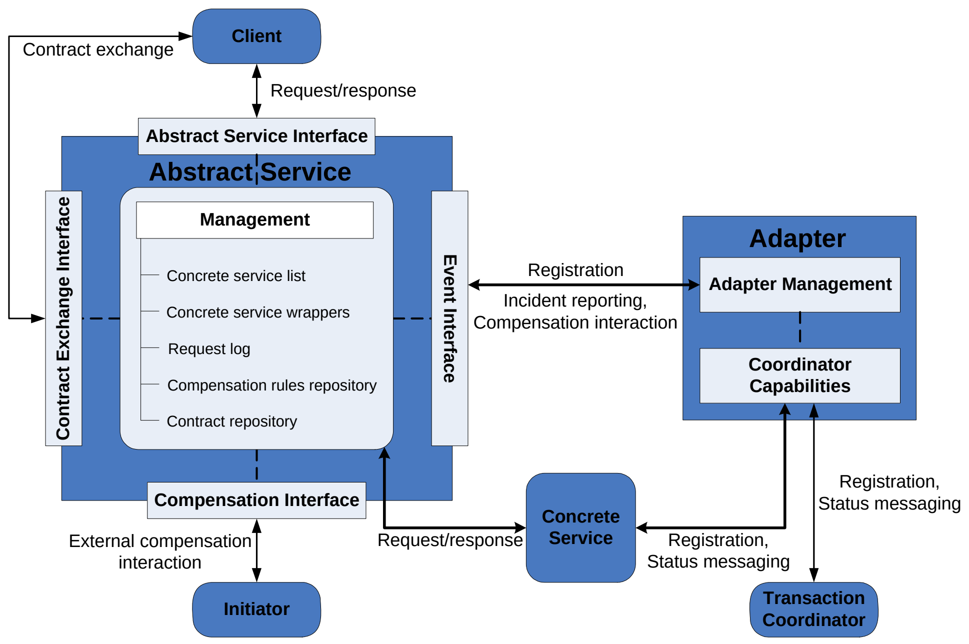

Fig. 3. The abstract service and adapter transaction environment

Arjuna Technologies Ltd. et al. 2005; Arjuna Technologies Ltd. et al. 2005]. We extend the architecture and the infrastructure based on those specifications, so that it can handle internally and externally triggered compensations. Figure 3 depicts the extension to the transaction Web service environment, namely the *abstract service* and the *adapter* components. This extension does not change the way how client, transaction coordinators and Web service providers operate. Instead of invoking a normal Web service, a client invokes an abstract service, which looks like a standard Web service to the outside. However, the abstract service is a management component for forward recovery compensation handling, which wraps multiple *concrete services* that offer the same functionalities and can thus replace each other. The abstract service is therefore a mediator between a client and the concrete service that performs the required operations. At the same time, the adapter component functions as a mediator between transaction coordinator, abstract service and concrete service to ensure proper transactional context and to provide the means to intercept failure notifications and create additional messages required in the compensation handling process.

3.1 Abstract Service

The central element of the extension is the notion of the *abstract service*. The client stub communicates with the Web service provider stub through the abstract service. An abstract service does not directly implement any operations, but rather functions as a management unit, which allows to:

—define a list of Web services which implement the required capabilities,

—invoke a service from the list in order to process requests which are sent to the abstract service,

—replace a failed service with another one from the list without a failure of the transaction, and

—process externally triggered compensations on the running transaction.

Distributed applications consisting of collaborating Web services have the advantage that normally single operations can be performed by multiple services from different providers. Which service will be chosen depends usually on the quality of service (QoS) requirements of the distributed application. The abstract service takes advantage of the existing diversity. To the outside, it provides an abstract interface and can be used like any other Web service, and uses the same mechanisms like SOAP [Nielsen et al. 2003] and WSDL [Christensen et al. 2001]. On the inside, it manages a list of Web services (called *concrete services*) which provide the required capabilities. When the abstract service receives a request, it chooses one of these services and invokes it. Which concrete service is chosen depends on the abstract service's implementation. In the simplest case, the abstract service only selects the next concrete service on the list. However, it would be possible to give the abstract service the capability to dynamically assess each concrete service and to choose the one that optimizes the client's QoS requirements. Interface and data incompatibilities between the abstract interface and the interfaces of the concrete services are solved by predefined wrappers.

This approach has multiple benefits:

—Usually, a client does not care which specific service handles his requests, as long as the job will be done successfully and in accordance with the contract. The abstract service design supports this notion by providing the capabilities to separate the required abilities from the actual implementation.

—The available list of concrete services enables the abstract service to provide enhanced compensation possibilities.

—The definition of an abstract service can be done independently from the business process in which it will be used. It can therefore be reused in multiple applications without the need for changes. If a specific service implementation is no longer usable, then the business process does not have to be changed, as this is being managed in the abstract service.

Figure 3 depicts the basic structure of an abstract service. Four interfaces are supplied to the outside: The service operations for which the abstract service has been defined can be accessed via the *abstract service interface*. A contract can be exchanged or negotiated by using the *contract exchange interface*. Execution events of a service (e.g. a failure) can be signaled via the *event interface*. Compensations can be triggered from the outside using the *compensation interface*.

On the inside, the main component is the *management* unit, which receives and processes requests, selects and invokes concrete services, and handles compensations. In order to do so, it has several elements at its disposal:

—*Concrete service list*: Contains the details of all available concrete services.

—*Concrete service wrappers*: Define the mapping of the generic abstract service interface to the specific interface of each concrete service.

—*Request log*: Holds all requests of the current session.

—*Compensation rules repository*: Manages the rules that control the compensation handling process.

—*Contract repository*: Contains the existing contracts with the different clients.

3.2 Adapter

Abstract services could be used in conjunction with a wide variety of technologies. Therefore, it would be preferable if the definition of the abstract service itself could be generic. However, the participation in a transaction requires capabilities that are different for each transaction management specification.

That is why the transaction specific requirements are encapsulated in a so-called *adapter* (see Figure 3). An abstract service registers with this adapter, which in turn registers with the transaction coordinator. To the coordinator it appears as if the abstract service itself has registered and sends the status messages. When the abstract service invokes a concrete service, it forwards the information about the adapter, which functions as a coordinator for the service. The service registers accordingly at the adapter as a participant in the transaction.

As can be seen, the adapter works as a mediator between the abstract service, the concrete service, and the transaction coordinator. The adapter receives all status messages from the concrete service and is thus able to process them before they reach the actual coordinator. Normal status messages can be forwarded directly to the coordinator, while failure messages can initiate the internal compensation handling through the abstract service.

If the adapter receives such an error message, it informs the abstract service, which can then assess the possibility of compensation. The adapter will then be informed about the decision, and can act accordingly. If for example the replacement of a failed concrete service is possible, then the adapter will deregister this service and wait for the replacement to register. In this case, the failure message will not be forwarded to the transaction coordinator. The compensation assessment could of course also show that a compensation is not possible (or desirable). In such a case, the adapter will simply forward the failure message to the coordinator, which will subsequently initiate the abort of the transaction.

3.3 Compensation Specifications

The compensation process on the side of the abstract service is controlled by rules, which specify when and how a compensation can be performed. These rules will usually be defined by the provider of the abstract service, who normally has the most knowledge about possible compensations. Two different kinds of compensations can be specified within these rules: Internally triggered compensations (arising from internal errors), which can be handled through a concrete service replacement operation, and externally triggered compensations. An example for an externally triggered compensation could be the handling of the mistake spotted by an administrator as described in the motivation scenario.

Each rule specifies the exact operations that have to be performed in the compen-

Nr	Compensation Type		Compensation Activities									
			ServiceReplacement	LastRequestRepetition	PartialRequestRepetition	AllRequestRepetition	CompensationForwarding	AdditionalServiceInvocation	AdditionalRequestGeneration	ServiceAbortInitiation	RequestSequenceChange	ResultResending
01		NoCompensation										
02	Internal	Repetition		■								
03	Internal	Repetition			■							■
04	Internal	Replacement	■	■								
05	Internal	Replacement	■		■							■
06	Internal	Replacement	■			■						■
07	External	Forwarding	▨	▨	▨	▨	■	▨	▨	▨	▨	▨
08	External	AdditionalService						■				
09	External	AdditionalRequest							■			
10	External	SessionRestart				■				■	■	■

■ **Included compensation activity** ▨ **Possibly included compensation activity**

Fig. 4. The compensation types and their included activities

sation process. For the purpose of defining the available compensation operations, we distinguish between basic *compensation activities*, which constitute the available single compensation operations, and complex *compensation types*, which are composed compensation processes consisting of multiple activities. This is shown in Figure 4. The reason for differentiating between compensation activities and types is to provide the means for *flexible* compensations without loosing control over the process within the abstract service environment. Moreover, while it would of course be possible to define simpler rules for external and internal compensations that do not offer many options, this approach would restrict the environment with regard to future extensions and new compensation strategies.

The compensation types specify which *combinations* of compensation activities can be defined in rules for handling internal and external compensations, as it is not desirable to allow every possible combination within the environment. When an abstract service receives a request for an internal or external compensation, it will first of all check whether a rule for the current situation exists, and if this is the case, it will validate each rule before executing the given set of compensation activities in order to guarantee that they are consistent with the available compensation types.

Therefore, although the combination of different compensation activities allows the definition of flexible and complex rules, it is not permitted to define arbitrary compensation handling processes. Only the predefined compensation types can be used, and it is thus guaranteed that an abstract service does not execute a process defined in a compensation rule that is not permitted or possible. At the same time, this approach allows the future extension of the environment with new compensation strategies: In order to test or include new compensation strategies, it is possible to simply define a new compensation type and extend the abstract service to accept it.

3.3.1 *Basic Compensation Activities.* Compensation activities are the basic operations which can be used in a compensation process. *ServiceReplacement* replaces the currently used Web service with a different one, which can offer the same capabilities and can thus act as a replacement. *LastRequestRepetition* resends the last request to the Web service. *PartialRequestRepetition* resends the last n requests from the request sequence of the current session (i.e. within the current transaction) to the Web service, while *AllRequestRepetition* resends all requests. *Compensation-Forwarding* forwards the external compensation request to a different component, which will handle it. *AdditionalServiceInvocation* invokes an additional (external or internal) service, which performs a particular operation that is important for the compensation (e.g. the invocation of a logging service, which collects data about a specific kind of compensation). *AdditionalRequestGeneration* creates and sends an additional request to the Web service. Such a request is not influenced by the client, and the result will not be forwarded to the client. *ServiceAbortInitiation* cancels the operations on the Web service, i.e. the service aborts and reverses all operations which have been performed so far. *RequestSequenceChange* performs changes in the sequence of requests that have already been sent to the Web service. *ResultResending* sends new results for old requests, which have already returned results.

3.3.2 *Compensation Types.* Compensation types aggregate multiple compensation activities, and thus form complex compensation operations, as shown in Figure 4. These types are the compensation actions which can be used for internal and external compensations, and which form the basis of the compensation specification language. There are currently 7 different compensation types.

The most simple type is *NoCompensation*, which does not perform any operation. If a Web service fails, then this will be signaled to the transaction coordinator, which will initiate the transaction abort.

The *Repetition* type is important for the internal error handling, as it repeats the last request or the last n requests. The last request can for example be resent to a Web service after a response was not received within a timeout period. A partial resend of n requests can for instance be necessary if the request which failed was part of a sequence, which has to be completely repeated after the failure of the final request. A partial repetition of requests will result in the resending of results for old requests to the client, which has to be able to process them.

The compensation type *Replacement* can be used if a Web service fails completely. It replaces the current service with a different one, and resends either all requests, a part of the requests, or only the last one. Resending only the last request is possible if a different instance of the service that has failed can be used as replacement, which works on the same local data and can therefore simply continue with the operations.

Forwarding is special in comparison with the other types, as it only indirectly uses the available activities. It forwards the handling of the compensation to a different component, which can potentially use each one of the compensation activities (which are therefore marked as "possibly included") in the process.

In an externally triggered compensation, it is sometimes necessary to invoke additional services and send additional requests to the concrete service. For this purpose, the compensation types *AdditionalService* and *AdditionalRequest* exist.

The final compensation type is *SessionRestart.* This operation is required if the external compensation request can not be handled without a restart of the complete session, i.e. the service has to be aborted and subsequently the complete request sequence has to be resent. The requested change will be realised by a change in the request sequence prior to the resending.

3.3.3 *Compensation Protocol.* While the compensation rules specify when and how a compensation can be performed, the compensation protocol controls the external compensation process itself and its interaction with the different participants.

An externally triggered compensation always has the purpose of changing one particular request that has already been processed at the service. More specifically, the compensation request contains the original request with its data that has to be changed (`request1(data1)`), and the new request-data (`data2`) to which the original request has to be changed to (`request1(data2)`). The participants in the protocol are the *abstract service*, the *client* which uses the abstract service in its business process, the *initiator* which triggers the external compensation (either the client itself, or any other authorized source like an administrator), the *concrete service* which is currently being utilized by the abstract service, and the *transaction coordinator.* An externally triggered compensation can only be performed if the transaction in which the abstract service participates has not yet finished, as this usually has consequences for the client due to result resending.

The protocol consists of two stages. The first stage is the *compensation assessment*: As soon as the abstract service receives a request for a compensation, it checks whether it is feasible and what the costs would be. To that end, predefined compensation rules are being used, which consist of a *compensation condition* (defines when a compensation rule can be applied) and a *compensation plan* (defines the compensation actions that have to be performed). The second stage of the protocol is the *compensation execution*, which performs the actual compensation according to the plan. Whether this stage is actually reached depends on the initiator: After the assessment has been completed and has come to a positive conclusion, the initiator, based on this data, has to decide whether the compensation should be performed or not.

As the client and the initiator of an external compensation can differ, the protocol contains the means to inform the client about the compensation process. It also ensures that the current concrete service and the transaction coordinator are informed about the status of the external compensation, as it is possible that the concrete service's (and thus the abstract service's) state changes due to the external compensation. The concrete service has to enter a specific external compensation handling procedure state for this purpose. While the concrete service is in this state, it will wait for additional requests from the abstract service, and the coordi-

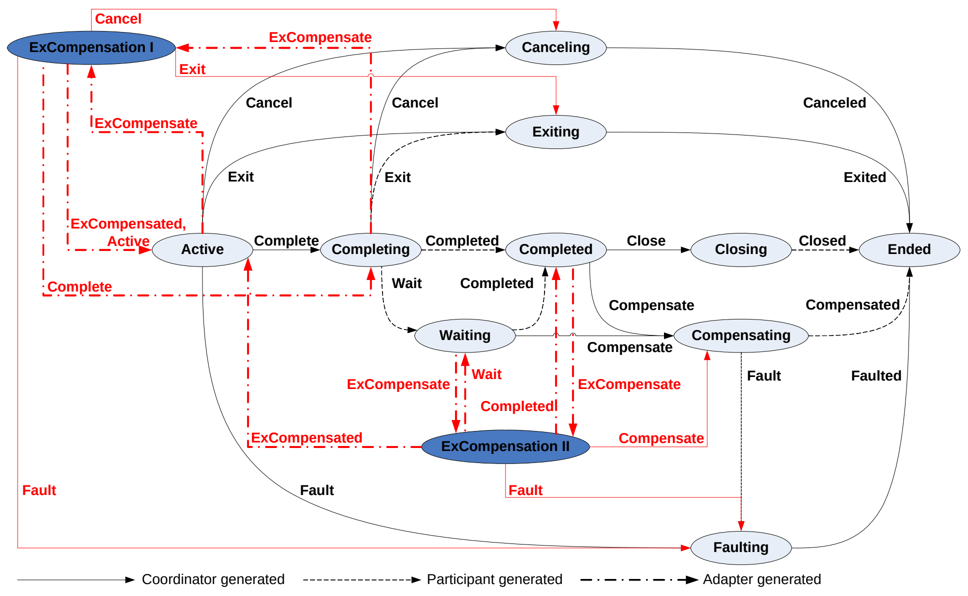

Fig. 5. The state diagram of the `BusinessAgreementWithCoordinatorCompletion` protocol with extensions for the external compensation handling

nator is not allowed to complete the transaction. While assessing the possibilities for a compensation, and while performing it, the abstract service can not process additional requests (and either has to store the requests in a queue, or has to reject them with an appropriate error message).

Because of the requirements of the compensation protocol, it is necessary to adapt the normal transaction protocol with additional state changes regarding the coordinator and participant (i.e. the concrete service). This has been done in our implementation for the `BusinessAgreementWithCoordinatorCompletion` protocol (refer to [Arjuna Technologies Ltd. et al. 2005]), using an extended version introduced in [Alrifai et al. 2006] as a basis, which uses transaction dependency graphs in order to solve cyclic dependencies. The result of the state diagram adaptation for the compensation protocol is depicted in Figure 5.

Two new states have been introduced, `ExCompensation I` and `ExCompensation II`. While both represent the external compensation handling procedure state which the concrete service has to enter, it is necessary to distinct between them, because, depending on the former state, different consequential transitions exist.

If the concrete service as participant is currently either in the `Active` state or the `Completing` state when receiving an `ExCompensate` notification from the adapter, it will enter the `ExCompensation I` state. While the concrete service is in this state, it will wait for new requests from the abstract service, and the coordinator will not finish the transaction. If the external compensation procedure is canceled after the assessment has been performed, the concrete service will be instructed to re-enter its former state by receiving either an `Active` or a `Complete` instruction from the adapter. The transaction processing can then continue in the normal way.

In contrast, if the external compensation is executed and performed successfully, the concrete service will receive an `ExCompensated` message, which instructs it to enter the `Active` state. This is necessary for two reasons: Firstly, because any additional requests as part of the external compensation handling require that the participant again performs the `Completing` operations. And secondly, because the abstract service's client will be informed about the external compensation that has been performed, and it is possible that additional operations are required by the client as a consequence of the compensation.

In addition to these options within the `ExCompensation I` state, the same transitions exist as in the `Active` and `Completing` states, i.e. the coordinator can `Cancel` the operations, and the participant can `Exit` or send a `Fault` notification.

If the concrete service is either in the `Waiting` or `Completed` state when receiving an `ExCompensate` message, it will enter the `ExCompensation II` state. In principle, the state has the same meaning as `ExCompensation I`: The concrete service will wait for new abstract service requests, and at the same time the coordinator is not allowed to finish the transaction. The concrete service will be notified to enter the `Active` state through an `ExCompensated` message after a successful external compensation execution. However, in contrast to `ExCompensation I`, different consequential transitions are available, and therefore it is necessary to separate these two states. In case of a compensation abort, the concrete service can be instructed to re-renter its former state through a `Wait` or `Completed` message. Moreover, a `Fault` message can be sent to signal an internal failure. Finally, the the coordinator can send a `Compensate` instruction while the concrete service is in the `ExCompensation II` state. The concrete service can only be instructed to `Compensate` if it is either in the `Waiting` or the `Completed` state. Therefore, it is necessary to introduce `ExCompensation II`, as this option is not available for the `Active` and `Completing` states and may thus not be permitted within `ExCompensation I`.

The extended state diagram contains new transitions generated by the adapter in addition to the ones from the participant (i.e. the concrete service) and the coordinator. This is actually a simplification, because although the adapter creates the messages and sends them to the coordinator and the participant, both are not aware of the fact that the adapter has sent them. To the coordinator it always looks as if the participant has sent the messages, while the participant thinks that the coordinator has sent them, as both are unaware of the extended transaction environment. Therefore, in order to obtain a state diagram that shows only transitions generated by either the coordinator or the participant, it would be necessary to create two different state diagrams, one from the participant's view and one from the coordinator's.

3.4 Application on the Client and Provider Side

The abstract service design can be applied on both, the client and the provider side. A client which wants to create a new distributed application using services provided by multiple providers can utilize abstract services in two different ways:

(1) The client can include the abstract service from a provider in its new business process, and can use the added capabilities.

(2) The client can define a new abstract service, which manages multiple concrete

services that can perform the same task.

The main goal of a Web service provider is a successful and stable execution of the client's requests in accordance with the contracts. If the service of a provider fails too often, he might face contractual penalties, or the client might change the provider. He can use abstract services in order to enhance the reliability and capability of his services by creating an abstract service which encapsulates multiple instances or versions of the same service. These can be used in case of errors to compensate the failure without the need for a transaction abort.

3.5 Client Contracts

While the multiple compensation capabilities of an abstract service allow the handling of internal and external compensations, it may not always be desirable for a client that these functionalities are applied. The abstract service environment therefore allows the definition and evaluation of *contracts*.

A client who wants to make use of the functionalities provided by an abstract service will negotiate a contract with the abstract service before sending the first request. This contract not only contains legal information and the Service Level Agreement, but can also specify which compensation operations the abstract service is permitted to apply. The abstract service dynamically adapts to this contract by checking the restrictions defined in it prior to performing a compensation: A compensation rule may only be applied if all necessary compensation operations are permitted via the contract. It can thus happen that although a compensation rule exists for handling a compensation, the abstract service will not apply it because the contract restricts the use of required compensation operations. Accordingly, an abstract service that is not allowed to use any compensation capabilities will act exactly like a standard Web service. A client therefore *can* make use of the forward recovery capabilities, but he does not *have* to, and thus always has the control over the environment's forward recovery compensation handling features.

Because of this ability to dynamically adapt to each client's contract, it is possible to use the same abstract service in a wide variety of distributed applications with differing requirements regarding compensation handling.

3.6 Transaction Environment Adaptation

The abstract service and adapter approach has been designed as an extension of the current transaction coordination structure so that it is easy to integrate it into existing environments and different transaction protocols. Therefore, it is not necessary to change either the client, coordinator or concrete service in order to use the internal compensation handling capability: An abstract service that manages different concrete services and that is able to replace failed concrete services can be used like a normal Web service and without any changes in the transaction protocol.

However, the introduced external compensation functionality for changing already processed requests requires some changes in the transaction environment:

(1) It is necessary to extend the existing transaction specification protocols to provide the capability to perform external compensations. This has been shown for the `BusinessAgreementWithCoordinatorCompletion` protocol in Section

3.3.3. Accordingly, the transaction coordinator and the participating Web services (i.e. concrete service) have to be able to handle this adapted protocol.

(2) The external compensation process requires that reports about a performed compensation and possibly the resending of results can be sent to the client of a transaction. It is therefore necessary that the client provides the expected interfaces and that he is able to process these reports in accordance with his business process.

The extend of the changes thus depends on the compensation handling requirements.

3.7 Middleware Prototype

The described design approach has been used in a prototype implementation. The implementation has been done using Apache Tomcat as Web container, and Apache Axis as SOAP engine. The WS-Transaction specification has been chosen for the transaction coordination, more specifically the adapted `BusinessAgreementWith-CoordinatorCompletion` protocol that has been introduced in Section 3.3.3. The implementation has been published online at SourceForge.net as the *FROGS* (**f**orward **r**ecovery c**o**mpensation handlin**g** **s**ystem) project:

`http://sourceforge.net/projects/frogs/`

4. EVALUATION

In this section we discuss various aspects of the middleware which we have evaluated. First of all, we look at exploitation of the proposed middleware in two experiments according to the motivating scenario we have discussed in Section 2. The experiments show how the components of the environment can work together in a concrete case, how compensation operates in such a context and how a concrete case with such a middleware is set up.

Secondly, we discuss an experiment which shows the performance of the proposed environment in comparison to the environment without forward compensation. The results of the experiments show that the proposed environment outperforms the standard environment based on backward recovery.

Thirdly, we propose an analytical evaluation model which carries out a cost benefit analysis of the proposed environment in contrast to the standard environment. We apply the analytical evaluation model to a concrete evaluation. By doing so we show under which conditions the environment with advanced forward compensations is beneficial.

Last but not least, we outline several possibilities for formal verification of the communication protocol we have introduced in this paper. We focus on one property to explain that there is always a computation which leads to compensation. Due to space limitation we do not show a complete proof.

4.1 Expoitation of the Middleware in Banking Scenario

We have experimented with the scenario introduced in Section 2 for test purposes. Multiple experiments have been performed within the implemented environment, two of which will be presented in this section.

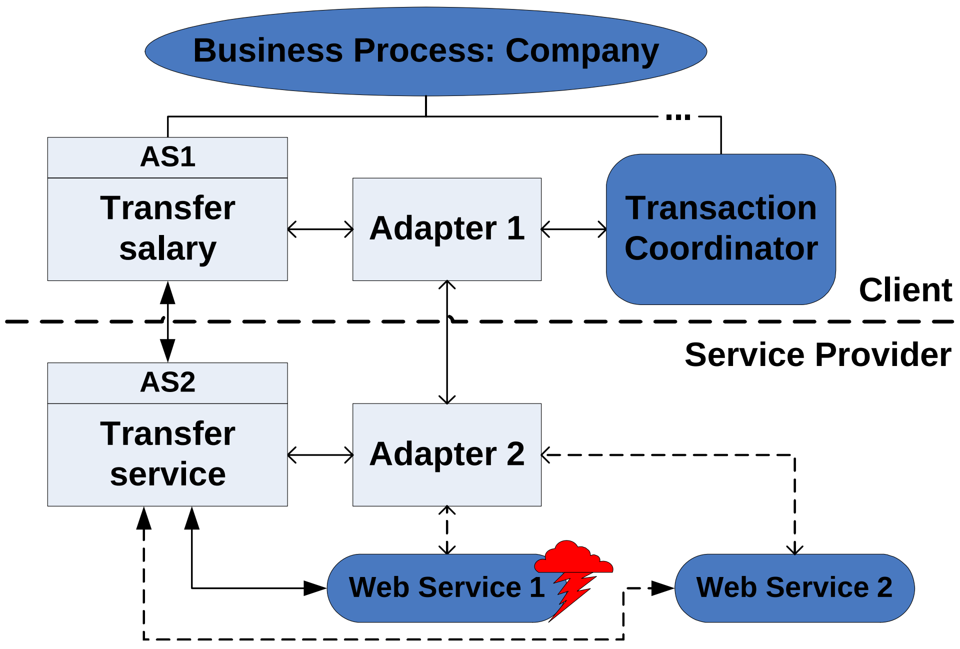

Fig. 6. Compensation on the provider side

The four services participating in the payment transaction have been realized as abstract services. The abstract services manage the standard Web services performing the required operations as concrete services.

The first experiment was devoted to the evaluation of the compensation of an internal service error. In this case, a failure of the concrete service on the provider side is simulated. Figure 6 shows the setup for the *transfer salary* operation: The abstract service *AS1* on the client side currently uses a concrete service that is itself an abstract service (*AS2*), which is operated by a service provider. The abstract service *AS2* uses *Web Service 1*, which performs the required operations. Figure 6 also depicts the interconnection of the services: *AS1* is registered as a participant at the *Transaction Coordinator* via *Adapter 1*, *AS2* is registered at *Adapter 1* via *Adapter 2*, and *Web Service 1* is registered at *Adapter 2*.

Now *Web Service 1* fails due to an internal error, and is thus not able to perform all operations required for the salary transfer. Instead of informing the transaction coordinator, abstract service *AS2* is informed, which assesses its compensation rules, the contract, and the available substitution services, and decides that a compensation is possible. *Web Service 1* is discarded and the request that failed is send to *Web Service 2*, which registers awith the adapter. *Web Service 2* is another instance of the same service, and can therefore simply continue with the request as it operates on the same local resources. This scenario shows that the signal of the service failure can be intercepted and the service replaced, without the need to cancel the complete transaction.

The second experiment evaluates an externally triggered compensation. Table I

Nr.	Transaction	Company (C)	Employee (E)	Tax (T)	Car Dealer (D)
		10.000	0	Y	Z
01	T1.debit(C,1.000)	9.000			
02	T1.credit(E,1.000)		1.000		
03	T1.debit(C,500)	8.500			
04	T1.credit(T,500)			Y+500	
05	T2.debit(E,150)		850		
06	T2.credit(D,150)				Z+150
		8.500	850	Y+500	Z+150

Table I. The transfer operations on the accounts in the scenario

Nr.	Transaction	Company (C)	Employee (E)	Tax (T)	Car Dealer (D)
...	...	...	...	...	...
07	**T1.debit(E,50)**		**800**		
08	**T1.credit(C,50)**	**8.550**			
		8.550	**800**	Y+500	Z+150

Table II. The additional operations on the accounts

summarizes the operations on the different accounts in the scenario described in Section 2. In this experiment, an administrator has found an error in the calculation of the salary: The company transferred 50 units too much to the account of the employee. The administrator directly sends a compensation request to the abstract service that handles the salary transfer (*AS1*). The abstract service assesses the request by consulting its compensation rules. In this scenario, the rules specify that this compensation is only allowed if the employee's account would still be in credit after the additional debit operation, in order to avoid the employee's account being in debit after the transaction.

The result of the assessment is positive, which is reported to the administrator, who can decide based on this data whether the compensation should be performed. He decides that the compensation is necessary. The abstract service compensates operations 01 and 02 from Table I by creating an additional debit and credit operation, as can be seen in Table II. The operations transfer 50 units from the employee's account back to the company's account, which thus compensates the initial problem. As an additional service, the abstract service initiates a precautionary phone call, which informs the employee about the change.

Subsequently, the compensation will be reported to the client, who has to assess whether any other services are affected according to its business process. It decides that the tax transfer does not have to be changed, while the payslip has to be updated, as the details of the salary have changed. The business process therefore initiates a compensation on the respective service, which handles this request by printing and mailing a new payslip. This shows that even the more complex initial problem could be solved without the need to abort the transaction.

These experiments have shown that the proposed design is successful in employing flexible compensation strategies in Web service transactions. It is thus possible to develop more robust distributed applications, where the abstract services are able to adapt their compensation rules to the contract they have with the client. Especially in long-running transactions, this approach helps to avoid unnecessary transaction aborts, and therefore saves money and time. While it is of course still possible that the abstract service itself encounters an error, it at least provides the capabilities to avoid transaction aborts due to concrete service failures. Moreover, it is possible to mix the new design with existing technology: The new capabilities can be used, but do not have to, as an abstract service can be employed like any other normal Web service.

The current implementation is a proof-of-concept of the proposed design architecture, and is still limited regarding certain aspects. The prototype of the abstract service uses only synchronous requests and does not allow parallel requests. Nevertheless, the same principles can be applied in this case, although additional request queue management will be required. Accordingly, the execution of compensation actions is currently performed only sequentially.

In this prototype implementation, compensation rules can be specified on the basis of an XML Schema definition. The following is an example of a rule for the processing of external compensation requests:

```
<cmp:ExternalCompensationRule identifier="refundSalaryDifference">
  <cmp:CompensationCondition>
    <cmp:RequestMethod identifier="transferSalaryMethod" />
    <cmp:ParticipantRequest identifier="getAccountBalanceMethod"
      parameterFactory="CheckEmployeeAccountParameterFactory">
      <cmp:Result resultEvaluator="AccountInCreditResultEvaluator" />
    </cmp:ParticipantRequest>
  </cmp:CompensationCondition>
  <cmp:CompensationPlan>
    <cmp:Compensation>
      <cmp:AdditionalRequest identifier="transferSalaryMethod"
        parameterFactory="RefundSalaryDifferenceParameterFactory" />
    </cmp:Compensation>
    <cmp:Compensation>
      <cmp:ServiceRequest
        serviceAddress="http://localhost:8080/axis/services/TelephoneCall"
        methodName="initializeTelephoneCall" />
    </cmp:Compensation>
  </cmp:CompensationPlan>
</cmp:ExternalCompensationRule>
```

The rule has been specified for the abstract service AS1, which handles the transfer of the salary in the scenario. The compensation condition consists of two single condition elements:

(1) `RequestMethod` - The rule applies for external compensation requests, which aim at changing requests that originally invoked the abstract service's method with the identifier `transferSalaryMethod`, i.e. it applies for external compensations that try to change the details of a salary transfer that has already been performed.

(2) `ParticipantRequest` - The second condition element specifies a request that has to be sent to the current concrete service. In this rule, the goal of the request is to check whether the account of the employee will still be in credit after the

excess amount that has been transferred has been refunded to the company's account. The condition's request invokes the concrete service's method that matches the abstract service's method with the identifier `getAccountBalanceMethod`. This method returns the current balance of the employee's account. The parameters of this request are created using the `CheckEmployeeAccountParameterFactory` *parameter factory*. This factory is a predefined class that implements a specific interface, and which is dynamically instantiated and invoked via the Java Reflection API. After the request has returned the current balance, the predefined `AccountInCreditResultEvaluator` *result evaluator* is responsible for checking whether the salary refund can be performed, and thus whether the rule's condition is fulfilled or not. Like a parameter factory, each result evaluator class has to implement a specific interface and is instantiated and invoked via the Java Reflection API.

The rule's compensation plan, which handles the salary refund, consists of two steps as well:

(1) `AdditionalRequest` - In the first step, an additional request is sent to the concrete service in order to perform the changes that are required, i.e. the transfer of the money back to the company's account. The request invokes the concrete service's method that matches the abstract service's method with the identifier `transferSalaryMethod`, which is the method that executes the salary transfer. The parameters for this request are again created by a predefined parameter factory `RefundSalaryDifferenceParameterFactory`.

(2) `ServiceRequest` - In the second step, an additional external service located at `http://localhost:8080/axis/services/TelephoneCall` is being utilized. The method `initializeTelephoneCall` has to be invoked, which does not require any parameters, and therefore no parameter factory has been specified. This external service performs the precautionary telephone call mentioned in the second experiment, which informs the employee about the error in the salary calculation and the refund that has been performed.

This example shows how compensation rules can be specified in the current implementation. By combining the different available condition and compensation action elements, it is thus possible to create complex rules for internal and external compensations. Although this specification of compensation rules is quite simple, it is sufficient for defining the different compensation strategies for the abstract service, and is flexible enough for creating and testing new compensation strategies within the extended transaction environment.

The use of predefined parameter factory and result evaluator classes makes the definition of rules simpler and reduces complexity, but at the same time of course externalizes condition and compensation action logic. This can make maintenance more difficult, because in this way the formal XML definition does not specify all parts of a compensation rule. Moreover, it is currently not possible to define AND/OR expressions for conditions. Each single condition element contained in a compensation rule therefore has to be fulfilled in order to fulfill the complete rule. In future implementations, it would be preferable to use a full-fledged rules language for the purpose of defining complex compensation rules. Nevertheless, the current

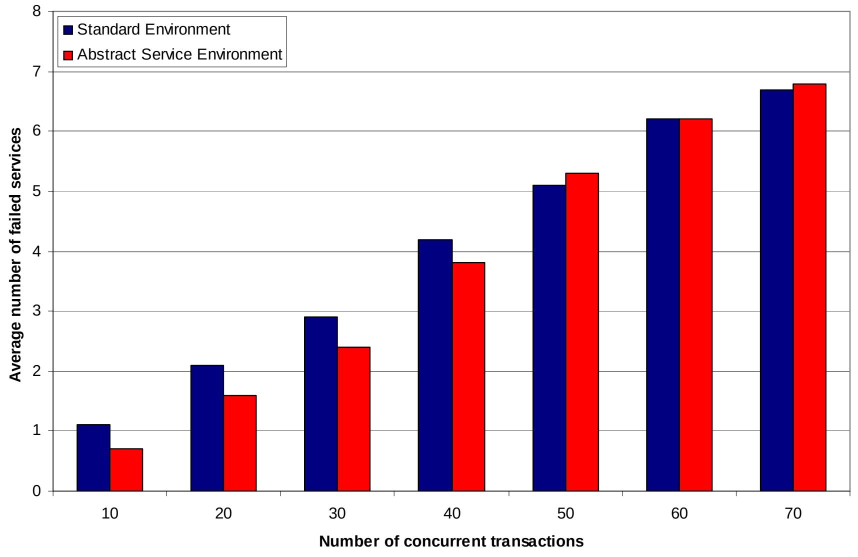

Fig. 7. The average number of failed services in the abstract service and the standard environment

rules specification is suitable for the prototype implementation, as it simplifies the creation of new rules as well as their analysis and execution. In addition, the design of the abstract service is flexible enough to allow the replacement of the compensation rules specification, in case this is necessary in future implementations.

4.2 Empirical Evaluation of Performance

In order to quantify the advantage of the proposed environment, an empirical evaluation has been performed. A scenario was set up in which multiple transactions concurrently invoke services, which have a predefined probability of encountering a failure. By gathering data within this scenario for both, the standard environment and the abstract service environment, it is possible to compare these two approaches with respect to the consequences of service failures.

In this scenario, each transaction always invokes two services:

(1) *Shared service* - A normal Web service that is invoked by all transactions in order to introduce transactional dependencies. This service exists only for this single purpose and will not fail when processing a request.
(2) *Individual service* - A service that will only be utilized by one transaction, and which has a failure probability of 10%. Whether or not a failure occurs is determined each time the service receives a request.

The transactions are limited to two participating services to allow a better comparison of the results. All services do not have any real functionality, which is not necessary for performing the tests, but instead simulate a required processing time of 3 to 20 seconds. When the scenario is being run for the standard environment, each individual service is a standard Web service. Accordingly, for the abstract

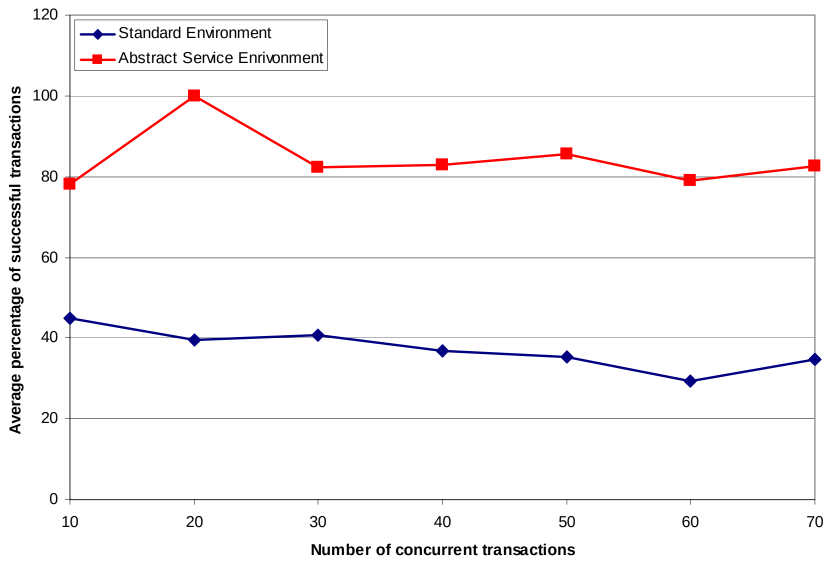

Fig. 8. The average percentage of successful transactions in the abstract service and the standard environment

service environment each individual service is an abstract service, and each manages two different concrete services. However, only the first concrete service carries the possibility of failure, the second one is the concrete service that can be used as a replacement in case the first one fails. Therefore, there will always be only one replacement operation per abstract service. All abstract services have a probability of 70 % of being able to perform a replacement in case of an error. This is to simulate that it is possible that no replacement service is available or that it is not allowed to perform a replacement according to the client contract.

A *client initializer* is responsible for creating and starting the clients within this scenario, each in an interval of 2 seconds. Each client will create a new transaction at the coordinator, and will then invoke first the shared service and then the individual service. The tests have been performed for a different number of concurrent transactions, and a new client is responsible for each new transaction.

The scenario has been repeated 10 times for each number of concurrent transactions, and the mean values for the gathered data have been calculated. Figure 7 depicts the average number of failed services in these tests, shown for 10, 20, ... 70 concurrent transactions in a comparison of both environments. As can be seen, the number of failed services is more or less the same, and therefore the consequences of these failures can directly be compared.

For each test run, the *percentage of successful transactions* and the *number of successful transactions per minute* have been calculated, and the average results for both environments are shown in Figure 8 and Figure 9, respectively.

Both, the number of successful transactions as well as the number of successful transactions per minute is considerably higher in the abstract service environment

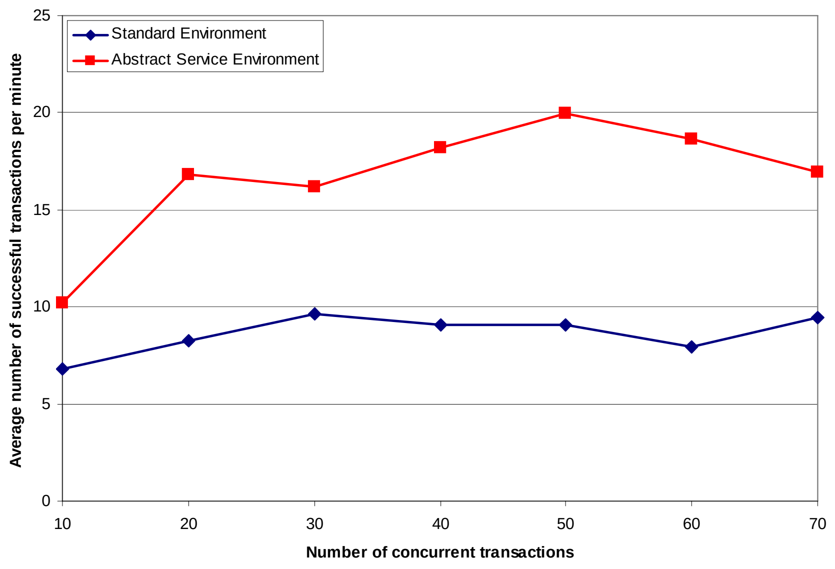

Fig. 9. The average number of successful transactions per minute in the abstract service and the standard environment

than in the standard environment. The abstract service environment is able to replace many failed services, and thus is able to avoid the rollback of dependent transactions. These results emphasise the advantage of the proposed design with respect to the overall success in an environment with concurrent transactions. Although the abstract service design increases the amount of messaging and the complexity of the environment, it is obviously beneficial because it decreases the number of transactions that have to be compensated and increases the reliability.

4.3 Analytical Cost-Benefit Evaluation

Evaluating the proposed environment is a challenging task, because a lot of different factors have to be taken into account when considering whether or not the abstract service environment should be applied. The main disadvantage of the approach is that it requires considerable message forwarding on the side of the abstract service and the adapter: The number of messages that have to be sent between client, participant, and coordinator is basically doubled. So when only the plain number of messages that have to be sent within a transaction is important or is considered, then the traditional environment is the better one. Figure 10 depicts this by comparing the required messages in the two different environments. For this comparison, a transaction is repeated consecutively 15 times, and in three of these transactions an error occurs. In order to handle this error, the standard environment aborts the whole transaction and restarts it, while the abstract service environment chooses a different direct replacement concrete service and continues with the request processing.

As can be seen, the number of messages that have to be sent in the abstract service

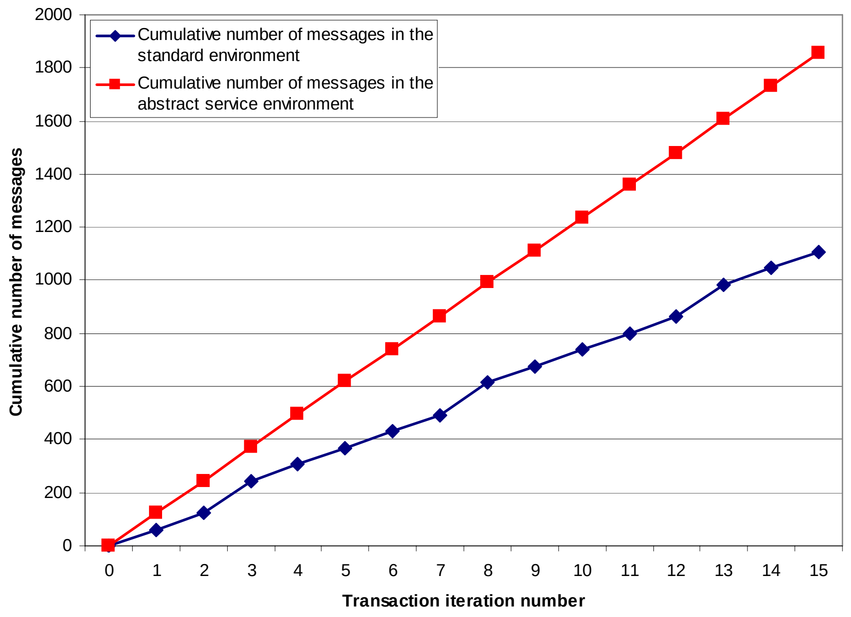

Fig. 10. Comparison of the cumulative message number progression in the standard and the abstract service environment

environment is considerably higher. However, the motivation for creating this new approach was to provide the means to avoid the *consequences* of a transaction abort, and not the reduction of the amount of messaging. The abort of a single transaction not only directly affects the service provider through possible contractual penalties (due to not fulfilling QoS promises) and loss of clients (*local effects*), but potentially can also trigger cascading transaction aborts, which can result in the loss of a huge amount of resources that have already been spent (*global effects*). Therefore, instead of only considering the number of messages that have to be sent, it is necessary to perform a cost-benefit analysis for the assessment of the proposed environment, which shows under which circumstances the extended environment is economically beneficial. For this purpose, it is required to calculate *net values* for both, the standard environment and the abstract service environment. A net value is defined as follows:

$$\textit{net value} = \textit{revenue} \text{ - } \textit{transaction costs} \text{ - } \textit{incurred losses}$$

Revenue is the sum of the earnings on the side of the participating Web services for performing the client's requests. *Transaction costs* are the costs that arise from the transaction management. Finally, *incurred losses* is the sum of all indirect costs of a transaction abort, i.e. contractual penalties, the costs for aborting cascading transaction, etc.

The *cumulative net value* of a transaction environment is the sum of all net values of the single transactions that are managed, and thus increases or decreases with

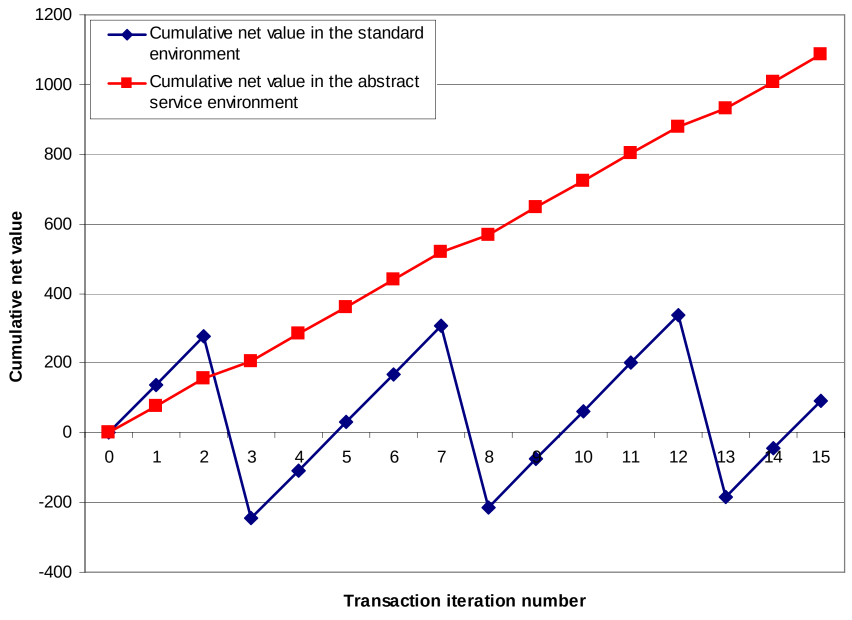

Fig. 11. Comparison of the cumulative net value progression in the standard and the abstract service environment

each new transaction. When considering this in the example from Figure Figure 10, it is possible to directly compare the sum of all 15 transaction net values for the standard environment with the abstract service environment. This is depicted in Figure 11.

The figure shows that the cumulative net value of the standard environment drops whenever a participating Web service encounters an error, mainly due to the incurred losses of the transaction abort and restart. However, the cumulative net value of the abstract service environment constantly increases because a transaction abort and its consequential losses could be avoided by using a different concrete service.

As can be seen, it is necessary to consider the whole picture of transaction management when assessing the proposed environment. A provider who considers using the abstract service approach in a service he wants to create has to decide on the following attributes of his new service:

—*Availability* - How available has the service functionality to be?

—*Reliability* - How reliable has the service to be, i.e. how often may it fail?

—*Adaptability* - How flexible does the service has to be with respect to request changes?

Moreover, he has to consider the costs that accumulate when a transaction abort is necessary as a result of a service failure or erroneous input data. The following factors have to be analyzed and taken into account:

—How many Web services participate in the transaction?

—How many dependent transactions exist per participant?

—How high is the probability of a participant failure?

—How high are the costs of performing a rollback on all dependent transactions?

—How high are these costs if the respective transactions have to be restarted?

—How many different clients are affected by the transaction aborts?

—How can the loss in reputation of the provider that offers the failed service be appraised if the service fails too often?

—How high are the contractual penalties that the provider incurs for a service failure/ for a reliability score that is too low?

—How often does it happen that a client enters incorrect information, which has to be changed?

As it can be seen, there are many different factors that have to be considered when assessing the use of the abstract service and adapter design within a given context. Usually, the service provider has only limited knowledge about the transaction in which one of his services participates. He only is informed about the number of transactions which currently use the service, and the contracts he has with the respective clients. Therefore, the provider's cost-benefit analysis will always be based on his limited knowledge and his estimations.

For the purpose of making a more general statement about the economic efficiency of the new design, we have created a simplified model which allows the calculation of the net value of both, the standard transaction environment and the enhanced transaction environment using abstract services. The two net values can be compared, and it is thus possible to decide whether the new design is economical. The model assumes that complete knowledge about all aspects of the transaction and all dependent transactions is available, and the different factors can be appraised. The goal of this model is not to have a realistic view on all aspects of the transactional environment, but rather to illustrate the methodology of the assessment.

The evaluation model will now be demonstrated on the basis of a simple scenario.

4.3.1 *Evaluation Set-Up.* In order to quantify and assess the abstract service design in comparison to the standard procedures with the proposed methodology, a simplified application case is being analysed: The cumulative costs are being assessed on the basis of the messages that have to be sent in the `BusinessAgreement-WithCoordinatorCompletion` protocol with atomic outcome as this was the context in which we performed the experiments presented above. Each message and thus each operation that it triggers has the same standard cost of "1". The analysis is therefore only valid for this specific transaction protocol and the scenarios that it assesses, but it gives an idea about the circumstances under which the application of the new design can be beneficial according to the net value analysis. Due to lack of space, the listing of the single messages that have to be sent is omitted and only the results are presented.

The transaction that is examined consists of p participating services, which receive s requests ($s \geq p$). In the standard environment, the normal processing of such a transaction without any failures requires

$$6p + 2s + 2$$

messages until all requests are processed and the transaction closed. However, if a failure occurs and the transaction has to be aborted and restarted,

$$12p + 4s + 1$$

messages are necessary until the transaction is closed.

In contrast to the standard environment, the abstract service environment requires for the normal transaction processing without any failures

$$12p + 4s + 2$$

messages, due to the doubled amount of required status messages. Should an error occur during the processing of the required requests, and a replacement of the failed concrete service be possible, then

$$12p + 4s + 2r + 8$$

messages have to be sent until the transaction is closed. The replacement of a concrete service requires that r requests are resent to the new one. If *r=1* then the new concrete service is a direct replacement, which operates on the same local resources and is thus able to directly continue with the processing of the request that caused the failure. If a replacement should not be possible, then the normal transaction abort and restart procedure has to be performed, which requires

$$24p + 8s$$

messages in the abstract service environment until all operations have been performed and the transaction is closed.

4.3.2 *Economic Model for Evaluation.* The net value of the *standard environment* considers the normal processing of the transaction without any operational errors, as well as the failure processing through a transaction abort. It is assumed that the distributed application still requires the operations to be performed, therefore a transaction restart is necessary subsequent to the transaction abort. In order to calculate the net value for this standard environment, the following parameters have to be provided:

—e : The revenue of the operations that are performed, i.e. the sum of all revenues of the participating services.

—p_N : The percentage of transactions in which no failures occur during the operations on the transactions' participants.

—c_{S1} : The costs of normal processing (i.e. without any errors), which includes for example costs for messaging and processing.

—c_{S2} : The costs of transaction processing in case a failure occurs. The abort and restart costs of the transaction after a failure include the costs of compensating the participants and resending the requests and thus reinvoking the different services.

—l_T : During a restart, the transaction has to be aborted by sending the according notifications to all participants. Therefore, all consequential costs that are incurred because of these actions have to be considered as well: Costs of cascading transaction aborts, contractual penalties because of service failures and aborts, the estimated customer migration, etc. These costs are summarized in this general loss value.

The net value v_S of the the standard environment can therefore be calculated as follows:

$$v_S = p_N \cdot (e - c_{S1}) + (1 - p_N) \cdot (e - c_{S2} - l_T) \tag{1}$$
$$e, c_{S1}, c_{S2}, l_T, p_N \in \mathbb{R}$$
$$0 \le p_N \le 1; e, c_{S1}, c_{S2}, l_T \ge 0$$

In order to simplify the model, it is assumed that the restart of the operations after a transaction abort is successful, and that no other errors occur.

In contrast to this, the net value of the *abstract service environment* considers the normal processing of the transaction without any operational errors, the replacement of a failed concrete service with a different one with request resending, as well as the failure processing through a transaction abort and restart if a replacement is not possible. In this model, it is assumed that all participating services are abstract services, and the performed replacement and/or restart does not encounter any additional errors. In order to calculate the net value for this environment, the following parameters have to be provided:

—e : As defined above.

—p_N : As defined above.

—p_R : The percentage of service replacement compensation operations, which can successfully be performed and thus handle a failure when it occurs.

—c_{A1} : The costs of normal transaction processing (i.e. without any errors) in the abstract service environment, which includes for example costs for messaging and processing.

—c_{A2} : The costs of transaction processing in case a failure occurs and a service replacement is possible. The costs of a concrete service replacement after a failure include the costs for assessing the possibility of a service replacement and for executing it.

—c_{A3} : The costs of transaction processing in case a failure occurs and a service replacement is not possible. The costs of the abort and restart of the transaction that can not be compensated through a service replacement include the costs for assessing the possibility of a service replacement, and the costs for compensating the participants and resending the requests to the different services.

—l_T : As defined above.

—l_R : If a service has to be replaced as failure handling, a loss value has to be considered, which appraises consequences of the single concrete service's failure.

The net value v_S of the the abstract service environment can therefore be calculated as follows:

$$v_A = p_N \cdot (e - c_{A1}) + p_R \cdot (e - c_{A2} - l_R) + (1 - p_N - p_R) \cdot (e - c_{A3} - l_T) \tag{2}$$

$$e, c_{A1}, c_{A2}, c_{A3}, l_T, l_R, p_N, p_R \in \mathbb{R}$$
$$0 \leq p_N \leq 1; 0 \leq p_R \leq 1; (p_N + p_R) \leq 1; e, c_{S1}, c_{S2}, c_{A3}, l_T, l_R \geq 0$$

The new design using abstract service and adapter components is economic, as long as the net value of the abstract service environment is higher than the net value of the standard environment:

$$v_A > v_S \tag{3}$$

$$\begin{gathered} p_N \cdot (e - c_{A1}) + p_R \cdot (e - c_{A2} - l_R) + (1 - p_N - p_R) \cdot (e - c_{A3} - l_T) \\ > \\ p_N \cdot (e - c_{S1}) + (1 - p_N) \cdot (e - c_{S2} - l_T) \end{gathered} \tag{4}$$

Using this simple model, it is possible to evaluate whether the utilization of the new design is beneficial by assigning the different required parameters according to experience or estimations, and then by comparing the net values of the standard and the extended transaction environment.

4.3.3 *Evaluation Results.* In order to demonstrate the utilization of the proposed model for calculating and comparing the net values, we assume that *p=5* and *s=15*. Moreover, we assume that all replacements are direct replacements (*r=1*). It is thus possible to calculate the basic costs for the model based on the assumption that each message costs "1" in operation:

$$\begin{aligned} c_{S1} &= 6 \cdot 5 + 2 \cdot 15 + 2 &&= 62 \\ c_{S2} &= 12 \cdot 5 + 4 \cdot 15 + 1 &&= 121 \\ c_{A1} &= 12 \cdot 5 + 4 \cdot 15 + 2 &&= 122 \\ c_{A2} &= 12 \cdot 5 + 4 \cdot 15 + 2 \cdot 1 + 8 &&= 130 \\ c_{A3} &= 24 \cdot 5 + 8 \cdot 15 &&= 240 \end{aligned}$$

In addition to these cost values, we assume that the following parameters have been estimated:

$$\begin{aligned} e &= 200 \\ p_N &= 85\% \\ p_R &= 10\% \\ l_T &= 600 \\ l_R &= 20 \end{aligned}$$

It is assumed that 85% of the transactions can be performed successfully and without any error in the first attempt. In the abstract service environment, 10% of the failed transactions can be compensated via a replacement of the current concrete service. The remaining 5% have to be compensated through a standard abort and restart of the transaction. The revenues of the operations in each transaction is set to 200, while the loss through consequences of a transaction abort are set to 600, and the loss through consequences of a replacement are set to 20.

Using these parameters, the following net values result from the calculation:

$$\begin{aligned} v_S &= 0,85 \cdot (200 - 62) + (1 - 0,85) \cdot (200 - 121 - 600) \\ &= 39,15 \end{aligned}$$

$$\begin{aligned} v_A &= 0,85 \cdot (200 - 122) + 0,10 \cdot (200 - 130 - 20) \\ &\quad + (1 - 0,85 - 0,10) \cdot (200 - 240 - 600) \\ &= 39,3 \end{aligned}$$

In this model and configuration, the abstract service design is economical, because its net value is higher than the standard environment's net value.

The functions of the net values for fixed revenue, cost, and loss parameters but for varying percentage values can be depicted as shown in Figure 12. As can be seen, the standard environment's net value constantly declines with the decreasing percentage of a failure-free transaction (p_N). In order to depict the net value of the abstract service environment, it is necessary to differentiate between its upper and lower boundary. The upper boundary defines the net value in case all failures can be compensated by replacing the failed concrete service. Accordingly, the lower boundary defines the net value in case replacing is never possible, and all failures have to be compensated by aborting and restarting the transaction. The area between these two boundaries is the range of possible net value functions for the abstract service environment, depending on the percentage of cases in which replacement operations can be performed in order to compensate the error.

It can be seen in Figure 12 that the abstract service environment is not economical for the specified parameters as long as the percentage of successful transaction without any failures is above 90%. However, this threshold of course changes dramatically as soon as higher losses are considered due to more or more expensive compensations in dependent transactions.

The proposed net value model also offers the capability to calculate how high the percentage of successful replacement compensation operations (p_R) has to be for a given percentage of successful transactions so that the abstract service environment is still beneficial:

$$p_R > \frac{p_N \cdot (-c_{S1} + c_{S2} + c_{A1} - c_{A3}) - c_{S2} + c_{A3}}{-c_{A2} - l_R + c_{A3} + l_T} \quad (5)$$

In the example calculation, the percentage of successful service replacement compensations has to be:

$$\begin{aligned} p_R &> \frac{0,85 \cdot (-62 + 121 + 122 - 240) - 121 + 240}{-130 - 20 + 240 + 600} \\ &> 9,98\% \end{aligned}$$

This minimum percentage can also be determined by using the graph shown in

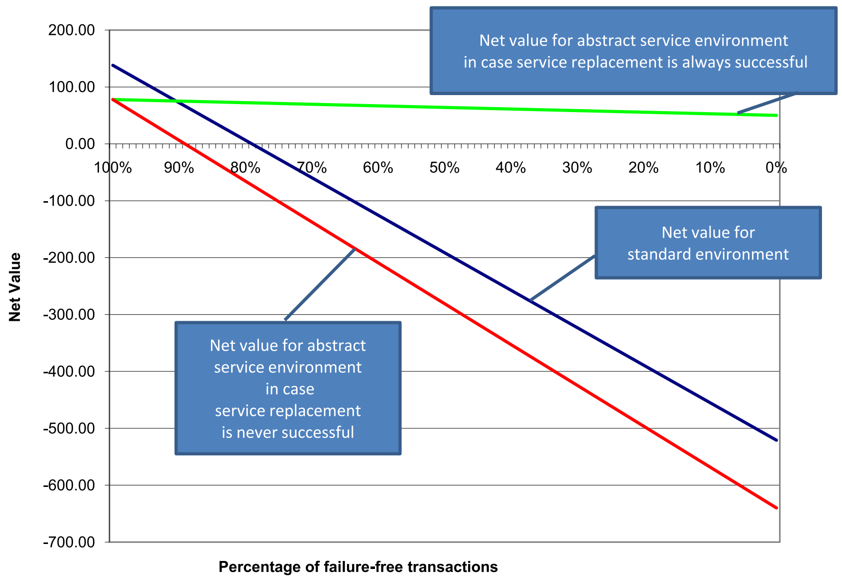

Fig. 12. Net value comparison between the standard and the abstract service environment

Figure 12. A specific abstract service environment net value function has to be included, which cuts the net value function of the standard environment at 85%. This is shown in Figure 13. By introducing a percentage scale between the upper and lower boundary of the abstract service environment's net value, it is possible to identify the percentage of compensation operations that have to be performed successfully by a service replacement instead of a transaction abort and restart. In this case, the percentage is approximately 65%, so 65% of the 15% of compensation operations have to be successful service replacements, which results in $p_R > 9,75\%$.

As can be seen, the proposed simplified model for the calculation and comparison of net values is both easy to apply and beneficial in the assessment of the new design.

4.4 Protocol Verification

Other means for the evaluation of our proposed environment exist, one of them is *protocol verification*. There are several possibilities how to do that and the decision which one to apply depends on which properties we would like to show. Time petri nets [Merlin and Farber 1976] or extended D-Time petri nets [Zuberek 1985] are suitable formalisms to perform model checking and formal analysis tasks to show various properties such as the recoverability of communication protocols. Other dialects of petri nets are suitable for this purpose as well.

Another possibility is to employ symbolic model checking to verify whether some properties of the protocol, which are specified in a dialect of a logic, hold. Suitable formalisms include but are not limited to linear temporal logic [Gerth et al. 1996] or branching temporal logic (CTL) [Clarke and Emerson 1982]. There are model checkers which support verification of properties stated in such a kind of logic such

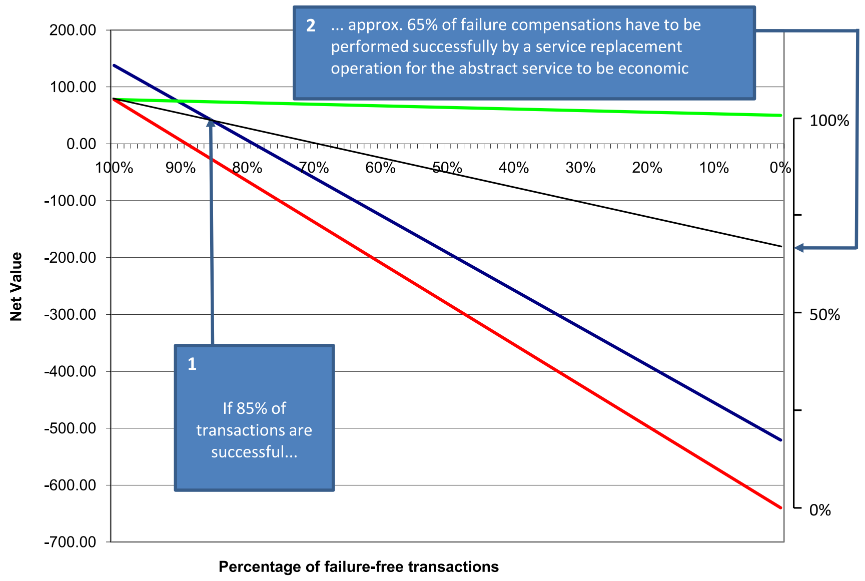

Fig. 13. The derivation of the minimum percentage of successful replacement compensation operations

as SMV [McMillan 1993]. Based on states and components in the environment and transitions defined in the protocol depicted in Figure 5, we can check the following property specified in CTL:

$\forall\Box(Active \rightarrow \exists\diamond ExCompensationI)$

This property specifies that there is a computation in which if a participant in the protocol from Figure 5 is in the `Active` state, it will eventually reach also the `ExCompensation I` state. By looking at the protocol, one can see that there is such a situation for example if an adapter as part of the environment sends an `ExCompensate` message. Other properties can be specified similarly.

It is not possible to show a complete formal model as well as the results of a formal analysis based on such a model in this paper due to space limitations. A CTL formalization and complete proofs are planned for a followup paper.

5. RELATED WORK

Compensations in transaction environments have been studied already in distributed database systems. Various advanced transaction models have been introduced [Elmagarmid 1992]. The main reason to go beyond the traditional transaction model was its unsuitability for long running transactions working on extended application areas, operating on more data items and resources spanning longer periods of time [Gray 1981]. Various advanced transaction models have been introduced where a notion of forward and backward recovery with compensations have been discussed such as Sagas [Garcia-Molina and Salem 1987; Garcia-Molina et al. 1991], nested transactions [Weikum and Schek 1991], split-transactions [Pu and Kaiser 1988], and others [Biliris et al. 1994; Barga and Pu 1995; Dayal et al. 1991]. Rule-based approaches for database extended transaction management have been introduced in [Günthör 1993; Klein 1991]. We have built on top of that work and proposed an environment for flexible Web service transactions. We employ a rule based approach in the abstract service component to specify compensations at the client as well as provider side. The abstract service component is separated the from coordination component in the environment, which enhances flexibility in compensation design without a need to exchange the standard coordination infrastructure. We allow for various transaction models at the provider side due to the introduced adapter component which hides this heterogeneity. Therefore our work provides an evidence how to realize in practice flexible compensations in autonomous Web service environments.

Forward recovery can be realized by using dynamic workflow changes, as described in [Reichert and Dadam 1998; Rinderle et al. 2006], which allow the semi-automatic adaptation of a workflow in case of errors. A change of the workflow process can for example consist of a deletion or jump instruction, or the insertion of a whole new process segment. The change can either be done on a running instance, or it can be performed on the scheme which controls the workflow, and which results in a change in all running instances. Refer to [Reichert et al. 2005] for details. Although this approach is very powerful, it has two major disadvantages. Firstly, it is in most cases only possible to perform these adaptations semi-automatically. Changing a workflow requires a lot of knowledge about the process and the current state it is in, and the implications a change would have. Therefore, it is often necessary for a human administrator to specify and control the change. Secondly, these kinds of workflow changes require a very strict definition of the process, including for example data and control links. Ad-hoc changes of business processes with normal orchestration languages like WS-BPEL (see [OASIS 2007]) is very difficult [Karastoyanova et al. 2005]. [Dobson 2006] provides a mechanism to overcome this difficulty through a compensation handler. Our approach provides a more flexible solution for compensations which is orthogonal to the business processes, concrete services, and transaction coordination.

Our compensation approach can be used in conjunction with the Enterprise Service Bus (ESB) [Chappell 2004], a powerful messaging infrastructure for business to business integration with Web services. The abstract service and adapter can be integrated through the ESB flexible extension mechanism. In this way, ESB can serve as a platform to exchange extended messages between business process, abstract services and adapters involved in the compensation conversation. Our approach can be used independently of ESB, employing ESB on top of the introduced infrastructure to integrate abstract services with workflow activities.

[Pires et al. 2003] introduces a notion of compensable Web services by specifying operations which can revert the execution. In our approach, we allow for a more complex specification of forward recovery compensations, which can be introduced at the client side, mediator side, as well as provider side. Two related approaches to a flexible compensation mechanism for business processes are proposed in [Yang and Liu 2006; Lin and Liu 2005]. In both cases, the focus is put on backward recovery. The compensation logic is treated as a part of coordination logic. In our approach, we separate the coordination from the compensation logic to provide for more flexibility.

Last but not least, our work is built on top of the preliminary paper on engineering compensations in web service environment [Schäfer et al. 2007]. In this paper we provide further technical details, as well as a discussion on the rule engine and the expressiveness of our rule based approach in contrast to practical needs together with an example. We also provide more details on the protocol implemented in the environment, and we introduce an extensive analytical evaluation together with an analytical cost-benefit model which have not appeared elsewhere. Furthermore, we consolidate other technical parts based on the experience we have gained from additional experiments and the evaluation.

6. CONCLUSIONS AND FURTHER WORK

We have described a new design approach for complex compensation strategies in current transaction standards. Two new components have been described, the *abstract service*, which manages replacement services and compensation rules, and the *adapter*, which separates the coordination protocol specific functions from the generic definition of the abstract service. We have also presented the protocol that handles the assessment and processing of externally triggered compensations. The design and the protocol have been successfully validated in a prototype implementation. Several experiments we have performed show that the environment with advanced compensations is beneficial. A net value model has been introduced, which can be utilized for the evaluation of the new abstract service environment in comparison to the standard environment, and thus to assess whether it is economical to apply the new design.

Regarding future work, we plan to run additional experiments with different compensation scenarios. Moreover, it will be necessary to further analyze the impact of the new compensation capabilities on the business process definitions. At the moment, it is only assumed that the business process is able to adapt to the signaled compensations. It will be required to analyze possible extensions of existing orchestration languages like BPEL in order to include the new capabilities. The current implementation will be extended to support the management of parallel request processing, and the definition of compensation rules will be adapted accordingly. We will also look on different properties and different formalisms to perform formal analysis of our protocol in our future work.

REFERENCES

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. 2003. *Web Services - Concepts, Architectures and Applications.* Springer.

Alrifai, M., Dolog, P., and Nejdl, W. 2006. Transactions concurrency control in web service environment. In *ECOWS '06: Proceedings of the European Conference on Web Services.* Zürich, Switzerland, IEEE Press Washington, DC, USA, 109–118.

Arjuna Technologies Ltd., BEA Systems, Hitachi Ltd., IBM Corporation, IONA Technologies, and Microsoft Corporation. 2005. Web Services Business Activity Framework. Published online at `ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf`.

Arjuna Technologies Ltd., BEA Systems, Hitachi Ltd., International Business Machines Corporation, IONA Technologies, and Microsoft Corporation. 2005. Web Services Coordination. Published online at `ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf`.

Arjuna Technologies Ltd., BEA Systems, Hitachi Ltd., International Business Machines Corporation, IONA Technologies, and Microsoft Corporation Inc. 2005. Web Services Atomic Transaction. Published online at `ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf`.

Barga, R. S. and Pu, C. 1995. A practical and modular implementation of extended transaction models. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases*, U. Dayal, P. M. D. Gray, and S. Nishio, Eds. Morgan Kaufmann, Zurich, Switzerland, 206–217.

Biliris, A., Dar, S., Gehani, N., Jagadish, H. V., and Ramamritham, K. 1994. ASSET: A system for supporting extended transactions. In *SIGMOD 1994: Intl. Conference on Management of Data*. ACM, Minneapolis, Minnesota, United States, 44–54.

Chappell, D. A. 2004. *Enterprise Service Bus*. O'Reilly Media, Inc.

Choi, S., Jang, H., Kim, H., Kim, J., Kim, S. M., Song, J., and Lee, Y.-J. 2005. Maintaining consistency under isolation relaxation of web services transactions. In *WISE*, A. H. H. Ngu, M. Kitsuregawa, E. J. Neuhold, J.-Y. Chung, and Q. Z. Sheng, Eds. Lecture Notes in Computer Science, vol. 3806. Springer, 245–257.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. 2001. Web Services Description Language (WSDL) 1.1. W3C note, W3C. March.

Clarke, E. M. and Emerson, E. A. 1982. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of Workshop Logic of Programs*. Lecture Notes in Computer Science, vol. 131. Springer, 52–71.

Dayal, U., Hsu, M., and Ladin, R. 1991. A transactional model for long-running activities. In *VLDB'1991: 17th International Conference on Very Large Data Bases*, G. M. Lohman, A. Sernadas, and R. Camps, Eds. Morgan Kaufmann, Barcelona, Spain, 113–122.

Dobson, G. 2006. Using WS-BPEL to implement software fault tolerance for web services. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*. Dresden, Germany, IEEE Press Washington, DC, USA, 126–133.

Elmagarmid, A. K., Ed. 1992. *Database transaction models for advanced applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. 1991. Modeling long-running activities as nested sagas. *IEEE Data Eng. Bull. 14,* 1, 14–18.

Garcia-Molina, H. and Salem, K. 1987. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. ACM Press, 249–259.

Gerth, R., Peled, D., Vardi, M. Y., and Wolper, P. 1996. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*. Chapman & Hall, Ltd., London, UK, UK, 3–18.

Gray, J. 1981. The transaction concept: virtues and limitations. In *VLDB 1981: Intl. Conference on Very Large Data Bases*. Cannes, France, 144–154.

Greenfield, P., Fekete, A., Jang, J., and Kuo, D. 2003. Compensation is not enough. In *7th International Enterprise Distributed Object Computing Conference (EDOC 2003)*. IEEE Computer Society, Brisbane, Australia, 232–239.

Günthör, R. 1993. Extended transaction processing based on dependency rules. In *RIDE-IMS'1993: Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*. IEEE, Vienna, Austria, 207–214.

Karastoyanova, D., Houspanossian, A., Cilia, M., Leymann, F., and Buchmann, A. P. 2005. Extending BPEL for run time adaptability. In *Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*. IEEE Computer Society, Enschede, The Netherlands, 15–26.

Klein, J. 1991. Advanced rule driven transaction management. In *Compcon Spring '91. Digest of Papers*. IEEE, San Francisco, CA, USA, 562–567.

Lin, L. and Liu, F. 2005. Compensation with dependency in web services composition. In *International Conference on Next Generation Web Services Practices (NWeSP 2005)*. IEEE Press, Seoul, KOREA, 183–188.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer.

MERLIN, P. AND FARBER, D. Sep 1976. Recoverability of communication protocols–implications of a theoretical study. *IEEE Transactions on Communications 24,* 9, 1036–1043.

MEYER, B. 1992. Applying "Design by Contract". *IEEE Computer 25,* 10, 40–51.

NIELSEN, H. F., MENDELSOHN, N., MOREAU, J. J., GUDGIN, M., AND HADLEY, M. 2003. SOAP version 1.2 part 1: messaging framework. W3C recommendation, W3C. June.

OASIS 2007. Web Services Business Process Execution Language Version 2.0. Published online at `http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf`.

PIRES, P. F., BENEVIDES, M. R., AND MATTOSO, M. 2003. Building reliable web services compositions. In *Web, Web-Services, and Database Systems: NODe 2002, Web- and Database-Related Workshops, Erfurt, Germany, October 7-10, 2002. Revised Papers*. LNCS, vol. 2593. Springer, Enschede, The Netherlands, 59–72.

PU, C. AND KAISER, G. E. 1988. Split-transactions for open-ended activities. In *Proceedings of the 14th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Bancilhon and DeWitt (Eds), Los Angeles.*

PULLUM, L. L. 2001. *Software Fault Tolerance — Techniques and Implementation*. Artech House, Inc., Norwood, MA, USA.

REICHERT, M. AND DADAM, P. 1998. ADEPTflex: supporting dynamic changes of workflow without loosing control. *Journal of Intelligent Information Systems 10,* 2, 93–129.

REICHERT, M., RINDERLE, S., KREHER, U., AND DADAM, P. 2005. Adaptive Process Management with ADEPT2. In *ICDE*. IEEE, 1113–1114.

RINDERLE, S., BASSIL, S., AND REICHERT, M. 2006. A Framework for Semantic Recovery Strategies in Case of Process Activity Failures. In *ICEIS*, Y. Manolopoulos, J. Filipe, P. Constantopoulos, and J. Cordeiro, Eds. 136–143.

SCHÄFER, M., DOLOG, P., AND NEJDL, W. 2007. Engineering compensations in web service environment. In *ICWE2007: International Conference on Web Engineering*, P. Fraternali, L. Baresi, and G.-J. Houben, Eds. LNCS, vol. 4607. Springer Verlag, Como, Italy, 32–46.

WEIKUM, G. AND SCHEK, H.-J. 1991. Multi-level transactions and open nested transactions. *IEEE Data Eng. Bull. 14,* 1, 60–64.

YANG, Z. AND LIU, C. 2006. Implementing a flexible compensation mechanism for business processes in web service environment. In *ICWS '06. Intl. Conference on Web Services*. IEEE Press, Salt Lake City, Utah, USA, 753–760.

ZUBEREK, W. M. 1985. Extended d-timed petri nets, timeouts, and analysis of communication protocols. In *ACM '85: Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective*. ACM, New York, NY, USA, 10–15.