Flypeer: A JXTA Implementation of DE Transactions

Amir Reza Razavi, Paulo R.C. Siqueira, Fabio K. Serra, and Paul J. Krause

[1] Department of Computing, School of Electronics and Physical Sciences, University of Surrey, Guildford, Surrey, GU2 7XH, UK
[2] Av. São Luiz, 86, cj 192, República, São Paulo - SP, Brasil
(a.razavi,p.krause)@surrey.ac.uk, (Paulo,Fabio)@ipti.org.br

Abstract. This paper introduces the Flypeer framework for transaction model in digital ecosystems. The framework tries to provide a fully distributed environment, which executes different type of order service compositions. The proposed framework is considered at the deployment level of SOA, rather than the realisation level, and is targeted to business transactions between collaborating SMEs as it respects the loose-coupling of the underlying services.

Introduction

The Digital Ecosystem is concerned with building an open source environment, through which businesses, in particular small to medium enterprises (SMEs), can interact within a reliable environment. The aim is to provide access to arbitrary services that help compose together to meet particular needs of the various partners. This collaborative software environment is being targeted primarily towards SMEs, who will be able to concatenate their offered services within service chains formulated on a digital ecosystem [1], [2].

Within a digital ecosystem a number of long-running multi-service transactions are expected to take place and therefore, of interest is the issue of providing support for long-term business transactions involving open communities of SMEs. A business transaction in this paradigm can be either a simple usage of a web service or a mixture of different levels of composition of several services from various service providers. We will argue that the current transaction and business coordination frameworks can lead to issues with tight coupling and violation of local autonomy for the participating SMEs.

Web services are the primary example of the Service Oriented Computing (SOC) paradigm [3]. The goal of SOC is to enable applications from different providers to be offered as services that can be used, composed, and coordinated in a loosely coupled manner. Web services in fact provide a realisation of SOC. Although recent years have seen significant growth in the use of Web Services, there are some very significant technological constraints that are stopping their full potential from being realised within a digital ecosystem.

The conventional definition of a transaction [4] is based on ACID properties. However, as it has been indicated [5], in advanced applications these properties can present unacceptable limitations and reduce performance [6].

The lack of consideration for the primary characteristics of SOC (such as loose-coupling) [7] or ignoring some important business requirements (such as partial results) [8], are important objections to the proposed transaction models, which also suffer from unnecessary complications of implementing a consistency model on top of the service-realisation boundary. In addition, the feasibility of a heavy coordinator framework causes some transaction models to use a centralised (or limited decentralised) coordination model [9].

The JXTA technology is a set of open protocols that enable any device on the network to communicate and collaborate in a P2P fashion, creating thereby a virtual network. It provides the peers, on this network, to share and discover resources, services and exchange messages with other peers, even if some of these peers are behind firewalls and network address translations (NATs). Thus, JXTA is an important key, in the Flypeer implementation, in the sense of give an effort only in the Transaction Model implementation.

In the first section, we have provided an overview for transactions in Digital Ecosystems; section 3 has explained the role of service oriented infrastructure of digital ecosystems for transaction model. We introduce the Flypeer framework in section 4 and section 5 provides the actual execution of transactions. Section 6 compares the model with similar models and the last section provides conclusion and clarifies future work and roadmap for the project.

Transactions in Digital Ecosystems

As we have already indicated, the very nature of business – as opposed to database – transactions, opens up a different angle from which to view transactions. For example, the specification of a transaction may involve a number of required services, from different providers, and allow it to be completed over a period of minutes, hours, or even days – hence, the term long-lived or long-running transaction. Indeed, a wide range of B2B transactions (business activities [10], [8]) have a long execution period. A business transaction between SMEs in a Digital Ecosystem can be either a simple usage of a web service or a mixture of different levels of composition of several services from various service providers.

It is important to stress that the long-term nature of execution frames the concept of a transaction in a digital ecosystem, since most usage scenarios involve long-running activities. In such cases, it is impractical, and in fact undesirable, to maintain full ACID properties throughout the lifetime of a long-running transaction. In particular, as we discussed in the Introduction, Atomicity and Isolation are questionable.

Within a digital ecosystem, a large number of distributed long-running transactions take place, each comprising an aggregation of activities which in turn involve the execution of the underlying service compositions. In such a highly dynamic environment there is an increased likelihood that some transaction (or one of its internal activities) will fail. This may be due to a variety of reasons (platform failure, service abort, temporary unavailability of a service, etc.) including the vulnerability of the network infrastructure itself (platform disconnection, traffic bottleneck, nodes joining or leaving the network, etc.).

The standard practice in the event of a failure is to trigger compensating actions that will effectively ‘undo’ the effects of the transaction – those effects visible before failure occurred. The objective is to bring the system to a state that is an acceptable approximation of the state it was before the transaction started. However, this is not a trivial task. Recovering the system in the event of a failure of a transaction (or an activity inside a transaction) needs to be done in a way that takes into account the dependencies both inside (to ensure all dependent activities are undone) but also outside the transaction (to ensure that any dependent activities of other transactions are also undone).

Further, and when considering SOA as the enabling technology for open e-business transactions, the recovery and compensation mechanism must respect the loosely- coupled nature of the connections, since interfering with the local state of the underlying service executions violates the primary requirement of SOA. In addition, access to the local state may not even be possible in a business environment with SMEs as service providers. This is an issue that has been largely ignored by current implementations of transaction models such as Web Services Transactions (WS-Tx) [11], [12] and Business Transaction Protocol (BTP) [13].

The abortion of a transaction, even if it is successfully recovered and compensated for, can be very costly in the business environment. Rolling back the whole system in the event of a failure may lead to chains of compensating activities, which are time-consuming and impact on network traffic as well as deteriorate the performance. For this reason it is important to build into the system capability or flexibility to deal with failure. In other words, it is key to design for failure by adding diversity into the system and allowing for alternative scenarios. The idea is to get some leverage in avoiding the abortion of a whole transaction (and all other dependent transactions) by means of allowing alternative paths of execution in cases where the path chosen originally encountered a failure.

It is also important to be able to preserve as much progress-to-date as possible. If an activity (sub-transaction) of a transaction fails, it is essential to undo (roll back) only those activities that have used results of this activity, i.e. are dependent on it. It is highly desirable to avoid rollback of other activities that have produced results (committed) and are not dependent on the failed activity. These are often referred to as omitted results and do no need to be undone as that would mean they will need to be re-done (re-started, re-calculated, re-computed) once the transaction is restarted (after abortion and recovery). Addressing omitted results can have significant benefits for SMEs in digital ecosystems in terms of saving valuable time and resources.

In earlier work, we have performed an extensive review of Transaction models, such as WS-Tx and BTP, that have been designed with web services in mind, and are currently widely used in practice [14]. Apart from certain issues regarding their coordination mechanism, which is geared towards centralised control, these models do not support partial results, do not provide capability for forward recovery, and there is no provision for covering omitted results.

Transactions and Service Composition

Based on the specification advocated in [15],[16],[17] service composition can be considered along the following dimensions: data, process, security, protocol. In this paper we are concerned with providing P2P network support for distributed transactions and hence we will be concerned with the aspects of data and process composition. In general, security and protocol compositions are usually addressed on top of the transactional layer.

In particular, process-oriented service composition is concerned with the following aspects:

Order: indicates whether the composition of services is serial or parallel.

Dependency: indicates whether there is any data or function dependency among the composed services.

Alternative service execution: indicates whether there is any alternative service in the service composition that can be invoked - alternative services can be tried in either a sequential or a parallel manner.

Following [15] these aspects can be seen within different types of service composition as follows.

Data-oriented service composition: The data generated at the service realisation level are released in terms of different data-objects. In this service composition, these data can be shared and manipulated between participants of a single transaction or, where partial results are concerned, be shared by participants of other transactions.

Sequential process-oriented service composition: This type of service composition invokes services sequentially. The execution of a component service is dependent on its previous service. These sequential dependencies can be based on commitment in which case we talk about Sequential with commitment dependency (SCD) where one cannot begin unless the previous service commits, or data in which case we talk about Sequential with data dependency (SDD) where one service relies on other service's outputs as its inputs.

Parallel process-oriented service composition: In this service composition, all the services can be executed in parallel. There may be data dependencies between them in which case we talk about Parallel with data dependency (PDD) or there may be differences in how and when the services can be committed (depending on some condition) in which case we talk about Parallel with commit dependency (PCD). When there are no dependencies between the parallel services we talk about Parallel without dependency (PND).

Alternative service composition: This type of service composition indicates that there are alternative services to be deployed and one of them is necessary. They are categorised to two different types: Sequential alternative composition (SAt) where there is an ordering for deployment of these services, and Parallel alternative composition (PAt), where there is no ordering (preference) between them and deployment of either service can satisfy the composition.

Generally, one or more service compositions can satisfy the user request. It can be seen that due to order and data in service composition, there can be increased complexity in composing services especially when transactions require a number of different services from different networked organisations. This means that there is a need for a context and data consistency model (at the deployment level) which can provide the correctness of the results.

Implementation Framework

The Flypeer project brings a few things to the table: web services technologies, long running transaction support and peer-to-peer. To avoid having to re-inventing a lot of wheels, JXTA is used to provide the basis of the peer-to-peer architecture. With this in mind, the main goal of Flypeer is the supporting Long Running Transactions.

To achieve that, Flypeer abstract the communication details of JXTA from the service developer. Flypeer exposes an interface, *FlypeerBusinessService*, which the developer implements. This interface defines the features available to service developers and, more importantly, allow the framework to correctly plug it into the execution of a composition – more on that in the next section.

Deploying services in a peer-to-peer node, in Flypeer, is just a matter of creating a jar file with the proper service implementation classes and a configuration file, which follows the mechanism defined in the *ServiceLoader* class available in Java, version 6 and above.

The project also tries to be as close as possible to WS-* standards. This means, for example, that we use WSDL (actually a simplified version of it) for service description, and should support SOAP calls from the "external" world in a not distant future version. More than that, with each new version of the framework we are trying to get closer and closer to commonly used standards – allowing for new developers to use the project with a lighter learning curve.

Modelling and Executing a Transaction

In order to model a long running transaction, a service composition has to be created with the required parameters of its services and then the transaction can be executed. A class diagram of this process can be found in the appendix A.

Regarding the service composition, Flypeer supports three types, SCD, PCD and Alternative Service compositions (recall section 3). All of these can be mixed to create from simple to complex compositions, using an xml file or programmatically, in what we call Transaction Context.

The *TransactionFlow* is the class which represents the Transaction Context and where is created the transaction flow. A *GenericFlow* implementation must be added into it to begin the composition. This can be a *SequentialFlow*, a *ParallelFlow* or an *AlternativeFlow* object. A *GenericFlow* object can receive Service or/and another *GenericFlow* objects.

The *Service* class represents the service which will be called, and its dependencies. The dependencies are mapped using the *addDependency* method, passing the input name only, if it has no dependency with another service, or passing also the dependent service object that will receive the output of this service as its input.

After composing the transaction, all of the required service parameters must be added into the composition. This is done using a Map where the key is the input name and the value is the parameter value which has to be a *Serializable* object. These parameters are validated automatically, before the transaction is started, according to the input mapping described in the service WSDLs.

After that, the transaction can finally be started using a *TransactionInitiator* object, which is obtained through the *CommandSimplePeer* object. Then, all service coordinators of this composition are notified and receive the Transaction Flow and the parameters definition. The transaction starts with the initiator's coordinator calling the first service's one. After that, each service coordinator calls the next one, until the last service coordinator on the transaction is reached.

In the end of the transaction, the initiator receives the result in a callback object defined prior to transaction start. This called implements the *FlypeerResponseListener* interface.

One of the missing pieces of JXTA is the transaction support. This provides us a nice window of opportunity to contribute a new feature back to the community – which is likely to happen when Flypeer is fully implemented.

A Simple Scenario

Fig 1 shows an example of a transaction in a tree structure with four services composed by three different composition types.

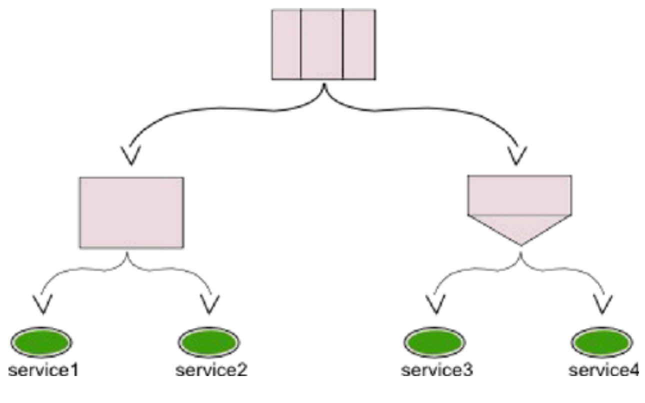


Fig. 1. Transaction in a tree structure

Fig 2 shows the transaction context (an xml file) for this example.

Executing a Transaction

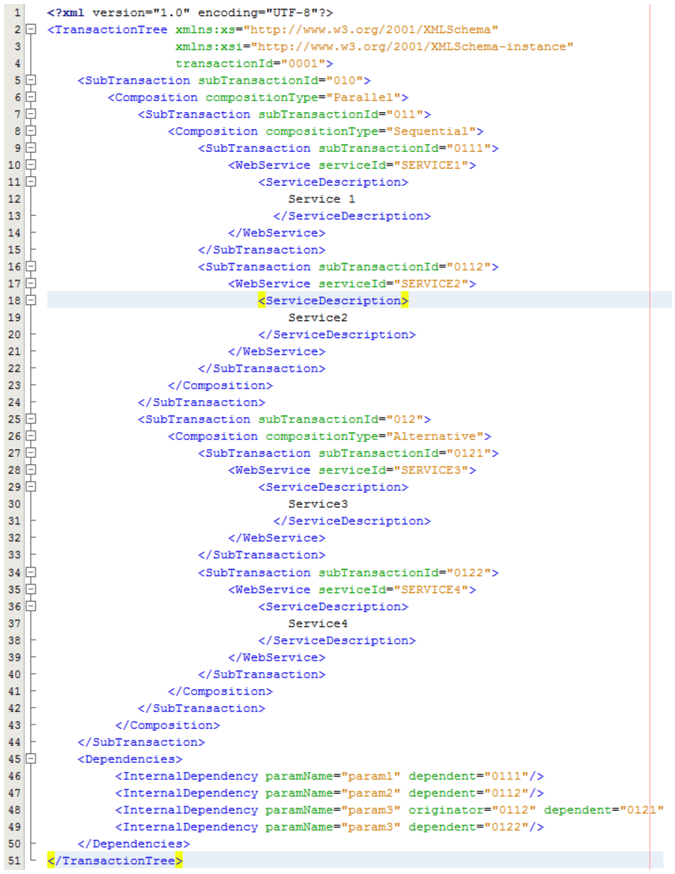
The transaction in Fig 1 can be modelled programmatically using the Flypeer Framework as it is demonstrated in Fig 3 overleaf.

First of all, there is a parallel composition which executes at the same time a sequential and an alternative composition.

The sequential composition runs first the Service1 and when it's finished the Service2 is called.

At this same time, the alternative composition also is being executed. First, the Service 3 is executed and if it runs with no errors, this composition is finished, but if something wrong happens the Service4 is called as an alternative for the Service3.

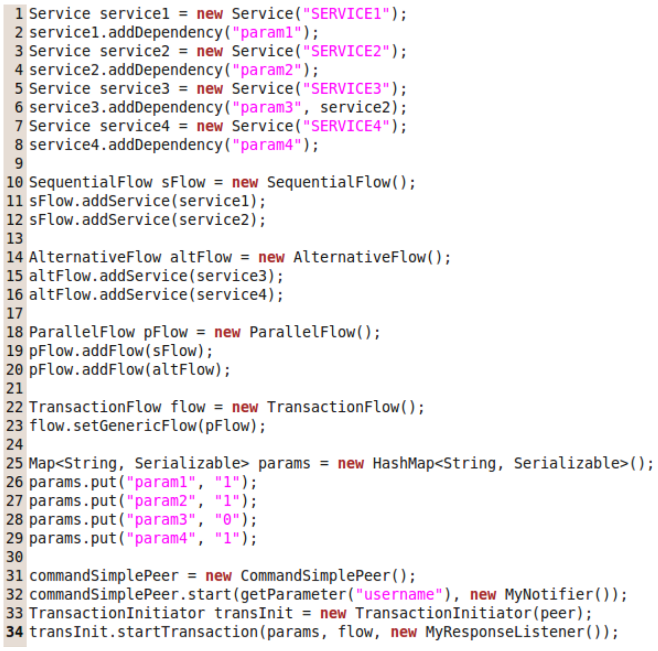
One thing to note about Service2 is that, as mapped in the Transaction Flow (Fig 2), the Service3 receives the output of the Service2 as an input parameter. So, in order to Service3 to be able to start its execution, Service2 has to have finished and sent its output to Service3 already.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<TransactionTree xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 transactionId="0001">
    <SubTransaction subTransactionId="010">
        <Composition compositionType="Parallel">
            <SubTransaction subTransactionId="011">
                <Composition compositionType="Sequential">
                    <SubTransaction subTransactionId="0111">
                        <WebService serviceId="SERVICE1">
                            <ServiceDescription>
                                Service 1
                              </ServiceDescription>
                        </WebService>
                    </SubTransaction>
                    <SubTransaction subTransactionId="0112">
                        <WebService serviceId="SERVICE2">
                            <ServiceDescription>
                                Service2
                            </ServiceDescription>
                        </WebService>
                    </SubTransaction>
                </Composition>
            </SubTransaction>
            <SubTransaction subTransactionId="012">
                <Composition compositionType="Alternative">
                    <SubTransaction subTransactionId="0121">
                        <WebService serviceId="SERVICE3">
                            <ServiceDescription>
                                Service3
                              </ServiceDescription>
                        </WebService>
                    </SubTransaction>
                    <SubTransaction subTransactionId="0122">
                        <WebService serviceId="SERVICE4">
                            <ServiceDescription>
                                Service4
                            </ServiceDescription>
                        </WebService>
                    </SubTransaction>
                </Composition>
            </SubTransaction>
         </Composition>
    </SubTransaction>
    <Dependencies>
         <InternalDependency paramName="param1" dependent="0111"/>
         <InternalDependency paramName="param2" dependent="0112"/>
         <InternalDependency paramName="param3" originator="0112" dependent="0121"
         <InternalDependency paramName="param3" dependent="0122"/>
    </Dependencies>
</TransactionTree>
```

Fig. 2. Transaction Context

When all compositions have finished the output is sent to the initiator as result of the transaction.



```java
Service service1 = new Service("SERVICE1");
service1.addDependency("param1");
Service service2 = new Service("SERVICE2");
service2.addDependency("param2");
Service service3 = new Service("SERVICE3");
service3.addDependency("param3", service2);
Service service4 = new Service("SERVICE4");
service4.addDependency("param4");

SequentialFlow sFlow = new SequentialFlow();
sFlow.addService(service1);
sFlow.addService(service2);

AlternativeFlow altFlow = new AlternativeFlow();
altFlow.addService(service3);
altFlow.addService(service4);

ParallelFlow pFlow = new ParallelFlow();
pFlow.addFlow(sFlow);
pFlow.addFlow(altFlow);

TransactionFlow flow = new TransactionFlow();
flow.setGenericFlow(pFlow);

Map<String, Serializable> params = new HashMap<String, Serializable>();
params.put("param1", "1");
params.put("param2", "1");
params.put("param3", "0");
params.put("param4", "1");

commandSimplePeer = new CommandSimplePeer();
commandSimplePeer.start(getParameter("username"), new MyNotifier());
TransactionInitiator transInit = new TransactionInitiator(peer);
transInit.startTransaction(params, flow, new MyResponseListener());
```

Fig. 3. Running the transaction

Comparison and Practicality

In Flypeer, the loosely coupled solution for a distributed transaction [18], [2] has been described in the context of digital ecosystems, and in particular we have been concerned with services, transactions, and network support within the digital ecosystem initiative. The structure of the interaction network within the architecture we have proposed emerges through the local interactions that take place in the context of long-running business transactions. In contrast with our model, WS protocols use a customised coordinator for avoiding inconsistency.

For providing a consistent environment, during concurrent actions (service deployment and compositions), WS-* (WS-AtomicTransactions and WS-BusinessActivity) and BTP, are using the two-phase commit (2PC) protocol, which requires synchronisation for the phases. This is applied through a centralised coordination framework, based on WS-Coordination [19]. Fig. 4 shows a simple example of the use of WS-Coordination for executing a transaction where the Initiator creates a coordination context and the Participants, based on their registered services, deploy their respective services. The synchronisation for concurrency control is done in a centralised manner. This causes a single point of failure as well as a single dependency on the provider(s) of the centralised coordinator framework.

In addition, a more careful study of this coordination framework, such as that reported in [9], shows it to suffer from some critical decisions about the internal build- up of the communicating parties - a view also supported in [20]. The Coordinator and Initiator roles are tightly-coupled and the Participant contains both

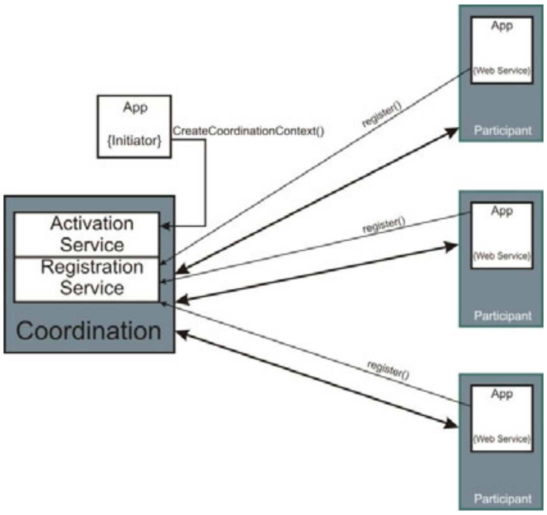


Fig. 4. WS-Coordination

business and transaction logic. These presumptions are against the primary requirements of SOA, particularly loose-coupling of services and local autonomy, and thus are not suitable for a digital business ecosystem, especially when SMEs are involved. This is because smaller organisations tend to be more sensitive in revealing their local design and implementation precisely because this is often where their model lies [2], [16].

Conclusions and Further Work

In this paper we have presented a framework for executing distributed long-term transactions in Digital Business EcoSystems. Various forms of service composition have been considered in order to provide a closer representation of business transactions within a service-oriented architecture. Our model is considered at the deployment level of SOA while respecting the loose-coupling of the underlying services. Further, it addresses omitted results, through forward recovery, in a way that does not break local autonomy.

Another strength of our approach is that it provides a light distributed coordinator which does not require to violate the autonomy of the local platforms or making any presumptions at the realisation level. This is particularly important in a business environment, especially for SME's. Further, these behaviour patterns can be applied dynamically on transactions in our approach, as was partly demonstrated in our example.

The work on progress of Flypeer will support the previous locking mechanism of the digital ecosystems transaction model (purposed in [8]). This is something we are keen to explore further as it lays the groundwork for a coordinated and collaborative service invocation specification to support long-term and multi-service transactions in Digital Ecosystems. The other part of Flypeer roadmap will provide a 'Trust framework' for businesses in digital ecosystems.

Acknowledgement

This work was supported by the EU-FP6 funded project OPAALS Contract No. 034824.

References

[1] Razavi, A.R., Krause, P., Moschoyiannis, S.: Deliverable D24.5: DBE Distributed Transaction Model (2006)
[2] Razavi, A.R.: Digital Ecosystems, A Distributed Service Oriented Approach for Business Transactions. PhD Thesis, University of Surrey (2009)
[3] Papazoglou, M.P.: Service-Oriented Computing: Concepts, Characteristics and Directions. In: Proceedings of the Fourth International Conference on Web Information Systems Engineering, December 2003. IEEE Computer Society Press, Washington (2003)
[4] Date, C.: An Introduction to Data Base Systems. Pearson Education, London (2003)
[5] Razavi, A.R., Moschoyiannis, S., Krause, P.: Preliminary Architecture for Autopoietic P2P Network focusing on Hierarchical Super-Peers, Birth and Growth Models (2007)
[6] Elmagarmid, A.K.: Database transaction models for advanced applications. Morgan Kaufmann, San Francisco (1992)
[7] Razavi, A.R., Moschoyiannis, S., Krause, P.: A Coordination Model for Distributed Transactions in Digital Business EcoSystems. In: Digital Ecosystems and Technologies (DEST 2007). IEEE Computer Society Press, Los Alamitos (2007)
[8] Razavi, A.R., Moschoyiannis, S., Krause, P.: An open digital environment to support business ecosystems. Peer-to-Peer Networking and Applications Springer Journal (2009)
[9] Furnis, P., Green, A.: Choreology Ltd. Contribution to the OASIS WS-Tx Technical Committee relating to WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity (November 2005), http://www.oasis-open.org/committees/download.php/15808
[10] Cabrera, L., Copeland, G., Johnson, J., Langworthy, D.: Coordinating Web Services Activities with WS-Coordination, WS-Atomic Transaction, and WSBusinessActivity (January 2004)
[11] Cabrera, L., Copeland, G., Feingold, M., Freund, R., Freund, T., Joyce, S., Klein, J., Lang-Worthy, D., Little, M., Leymann, F.: Web Services Business Activity Framework (WS-BusinessActivity). In: 2005 IBM Developer Works (2005)
[12] Cabrera, L., Copeland, G., Feingold, M., Freund, R., Freund, T., Johnson, J., Joyce, S., Kaler, C., Klein, J., Langworthy, D.: Web Services Atomic Transaction (WS-AtomicTransaction). IBM, US (2005)
[13] Ceponkus, A., Dalal, S., Fletcher, T., Furniss, P., Green, A., Pope, B., Inferior, A.: Business transaction protocol V1.0. OASIS Committee Specification (June 3, 2002)
[14] Razavi, A.R., Krause, P.: Integrated autopoietic DE architecture - D3.10 (2009)
[15] Yang, J., Papazoglou, M.P., van den Heuvel, W.J.: Tackling the Challenges of Service Composition in e-Marketplaces. In: Proceedings of the 12th International Workshop on Research Issues on Data Engineering: Engineering E-Commerce/E-Business Systems (RIDE-2EC 2002), San Jose, CA, USA (2002)
[16] Singh, M.P., Huhns, M.N.: Service-Oriented Computing: Semantics, Processes, Agents. Wiley, Chichester (2005)
[17] Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., Kramer, B.J.: Service-oriented computing: A research roadmap. In: Service Oriented Computing, SOC (2006)

[18] Razavi, A., Moschoyiannis, S., Krause, P.: An open digital environment to support business ecosystems. In: Peer-to-Peer Networking and Applications. Springer, New York
[19] Cabrera, L., Copeland, G., Feingold, M., Freund, R., Freund, T., Johnson, J., Joyce, S., Kaler, C., Klein, J., Langworthy, D., Little, M., Nadalin, A., Newcomer, E., Orchard, D., Robinson, I., Shewchuk, J., Storey, T.: Web Services Coordination (WS-Coordination) (August 2005)
[20] Vogt, F.H., Zambrovski, S., Gruschko, B., Furniss, P., Green, A.: Implementing Web Service Protocols in SOA: WS-Coordination and WSBusinessActivity. In: CECW, vol. 5, pp. 21–28 (2005)