Long-running Transactions: semantics, schemas, implementation

Sotiris Moschouyiannis, Member, IEEE, Amir R Razavi, Yongyan Zheng, Paul Krause

Department of Computing, Faculty of Engineering and Physical Sciences, University of Surrey, Guildford, Surrey, GU2 7XH, UK

E-mail: {s.moschouyiannis | a.razavi | y.zheng | p.krause}@surrey.ac.uk

Abstract—In this paper we describe a formal model for the distributed coordination of long-running transactions in a Digital Ecosystem for business, involving Small and Medium Enterprises (SMEs). The proposed non-interleaving model of interaction-based service composition allows for communication between internal activities of transactions. The formal semantics of the various modes of service composition are represented by standard XML schemas. The current implementation framework uses suitable asynchronous message passing techniques and reflects the design decisions of the proposed model for distributed transactions in digital ecosystems.

Index Terms—service orchestration, concurrency, vector semantics, schema, message passing.

I. INTRODUCTION

The concept of a Digital Ecosystem for business emerges as a key feature in the era of the Digital Economy. The importance of a pool of rare species in sustaining (innovation in) an ecosystem is well understood in the studies of biodiversity. In a digital ecosystem for business, and if we read sustainable economic growth for extinction, SMEs can be seen to play the role of ‘rare species’.

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. The specification of a business transaction may allow it to be completed over a period of minutes, hours, or even days – hence, the term long-lived or long-running transaction. The execution of a long-running transaction corresponds to conducting a business activity and typically comprises a number of subtransactions or activities that involve the execution of a number of underlying services.

This makes Service-Oriented Computing (SOC) [1], whose goal is to enable applications from different providers to be offered as services that can be used, composed and coordinated in a loosely-coupled manner, the prevalent computing paradigm in a digital business ecosystem. The actual architectural approach of SOC is called SOA [2] and is particularly applicable when multiple distributed applications are running on varied technologies and platforms need to communicate with each other.

Business transactions in a business-to-business (B2B) context typically involve interactions between multiple partners, either service providers, or service consumers, or both. This requires that all partners behave in a coordinated manner – partners must follow some protocol to execute a transaction effectively.

Conventional transaction models such as Sagas [3] or the more recent models targeting web services Web Services Transactions (WS-Tx) [4] and Business Transaction Protocol (BTP) [5] seem to be geared towards centralised control (based on the WS-Coordination framework [6]) which means some knowledge of the internal build-up of the participants is required. This violates the primary requirement of SOA for loosely-coupled services and may not be acceptable or even possible in a business environment – in a digital ecosystem involving communities of SMEs this raises a barrier for the adoption of SOA as it violates local autonomy.

Part of the problem seems to be that these frameworks lack a formal model for the coordination of the underlying services involved in the execution of a long-running transaction. Further consequences of this may come to view when a transaction needs to be compensated due to a subsequent failure. In this paper however, we will focus on describing the forward behaviour of a transaction, which concerns the coordination of the underlying services and will not consider compensation mechanisms, semantics and schemas (compensating behaviour). Preliminary ideas on recovery management that build on the ideas presented in this paper can be found in [17].

In this paper we give a formal description of long-running multi-service transactions in terms of the distributed orchestration of the underlying service executions. The formal semantics, in the first instance, is aimed at describing the behavioural patterns services should follow in order to increase confidence in a successful outcome. The coordination of a transaction in our approach amounts to interaction-based composition of the underlying services and thus we advocate the use of XML schemas that reflect the message-based protocol used in a transaction. Finally, we describe a prototype implementation of our ideas on the distributed coordination of long-running transactions and the associated structural schemas.

The remainder of this paper is structured as follows. Section II outlines earlier work on a distributed transaction model for digital ecosystems. Section III provides a standard schema which reflects the resulting transaction structure and can be used for deriving XML files that describe the transaction in question. Section IV introduces the formal description of a transaction in terms of its underlying service executions and shows how various modes of service
composition can be expressed within our formal model of a transaction. We also provide standard schemas that generate XML descriptions of the required service coordination, as prescribed by the formal model. The prototype implementation of the transaction structure schema is then presented in Section V. The paper finishes with some concluding remarks and ideas for future work given in Section VI.

II. DISTRIBUTED TRANSACTION MODEL FOR DES

In earlier work [7], we have described a distributed transaction model for networked organisations in the digital ecosystem paradigm. Here we outline the basic ideas.

It is often the case that internal activities of a transaction need to share results before the termination of the transaction (transaction commit). More generally, dependencies may exist between activities inside a transaction due to the required ordering on service invocations (e.g. book a hotel only after booking the flight) or due to the sharing of data (one service execution uses the results of another). The use of log structures, given in the form of directed graphs, has been described in [7], [8] for capturing the internal and external dependencies of a transaction.

For example if a transaction involves the sequential composition of services, this gives rise to a dependency (due to ordering and / or data) between the corresponding service invocations. As a consequence, if one is aborted for some reason, then the other also needs to be aborted in order to maintain consistency across the transaction. In [17] we have introduced the so-called Internal Dependency Graph (IDG) for representing such dependencies. Similarly, an External Dependency Graph (EDG) is used for representing dependencies between services from different transactions in a digital ecosystem for business. Fig. 1 shows the IDG for pairs of services that have been composed using different types of composition, namely Sequential with Data Dependency (SDD) and Parallel with Data Dependency (PDD).

The benefit of using these lightweight logs is that each platform coordinator only needs to have knowledge about its services and their dependencies to other coordinators’ services – it needs to know only what happens before and after its own services. Due to space limitations we omit further details.

In our model a transaction is represented by a tree structure that allows for nested subtransactions and exemplifies the local coordination that is required for the services involved to be performed in unison.

Drawing upon the latest work on an extended service-oriented architecture for a business environment [2], we have considered five different types of coordinators which allow for various modes of service interaction in our model. In the transaction tree of Fig. 2, the services $s_3$ and $s_4$ for example are children of a sequential coordinator and hence $s_4$ can only be executed after $s_3$. In other words the execution of $s_4$ is dependent on the (successful) execution of $s_3$, e.g. $s_3$ uses the results released by $s_3$.

The scenario described in Fig. 2 has appeared in [7] and has been simplified here, nevertheless the transaction in question is complicated enough to illustrate the key ideas of our formal modelling approach.

III. TRANSACTION CONTEXT SCHEMA

The DE nested transaction model can be represented in a context schema. This is used by the Initiator of a transaction to specify the different levels of compositions of the underlying services. The schema also shows the boundaries and requirements of the model and the need for a formal language to describe transactions, which will be given in Section IV. Fig. 3 shows such a schema (the Netbeans 5.5.1 source can be found following [9]).

The nested transaction structure is imposed in the schema using the ‘Coordinator’ element which can include a service composition (‘Composition’ element in the schema) or usage of a simple web service (notice the ‘WebService’ element in the schema), including a purely data-oriented web service (a leaf of the tree). Each service composition (‘Composition’) has a type (‘Composition-Type’), which determines the mode of the interaction-based composition, whether it is sequential, parallel or alternative, and we will see how each is treated formally in our approach in Section IV. It has two or more ‘Coordinator’ elements which introduce recursion - another service composition or a simple web service. This recursive definition can generate an XML representation of the transaction tree. In the following sections we interpret this context schema into a formal language which allows for an XML presentation of the semantics of a transaction which can then be implemented in different platforms.

![Fig. 1 IDG for sequential and parallel service compositions](image)

![Fig. 2 Transactions in a tree structure](image)
IV. A FORMAL LANGUAGE FOR TRANSACTIONS

In this section we introduce a formal language for describing long-running transactions. Apart from the dependencies between service executions, there is a high degree of concurrency in a transactional environment since real problems require a number of activities to take place in parallel. For this reason we will find the general theory of non-interleaving representation of parallel behaviour found in [10] of great use in what follows. As mentioned before, our objective is to get a thorough understanding of the behaviour the underlying service compositions need to exhibit for a successful outcome of the transaction as a whole.

The semantics is intended to describe the behaviour of a transaction in terms of its services at the deployment level, but not the low-level computations performed by the services themselves. Note that services in a digital ecosystem for business are offered from different service providers and it is important that we defer from interfering with the local state of the service execution. The adoption of a service-oriented architecture for distributed transactions reinforces our interest in all environmentally observable actions that take place during the course of the execution of a long-running transaction. This means it is appropriate to consider that any action within the transaction model has no significant duration, in the sense that (i) it either occurs as a whole or not at all; (ii) it occurs either wholly before, or wholly after, or wholly in parallel with, every other action.

A transaction may thus be associated with a finite set of events or significant events or actions that may occur during execution, e.g. service invocation, initialisation, commitment, service return, release result (return), termination, abort, etc. We denote this set of actions of a transaction by \( M \).

A transaction \( T \) is associated with a set of leaves or access points \( L \) which consists of a set of basic services \( S \), a set of data-oriented coordinators \( D \) and a set of delegation coordinators \( D_{tg} \). Hence, \( L = S \cup D \cup D_{tg} \). We further require that the sets \( S, D \) and \( D_{tg} \) are pairwise disjoint.

Actions within a long-running transaction take place on the leaves and therefore each leaf is in turn associated with a set of actions that may occur on that leaf, depending on its nature. We denote this set by \( \mu (l) \), for each leaf \( l \in L \), and require that \( \bigcup_{l \in L} \mu (l) \subseteq M \), so \( \mu \) is an indexed cover for the set \( M \).

As can be seen in Fig. 2 a transaction has a number of activation points, namely the leaves of the corresponding transaction tree. Thus, instead of modelling a transaction by a sequential process that would generate a trace of a single access point, we model the behaviour of a transaction by considering a set of such sequences at the same time, one sequence for each leaf. This draws upon Shields’ vector languages [10] and allows us to capture what is happening on each access point of a long-running transaction.

Transaction vectors. Let \( T \) be a transaction. We define \( V_T \) to be the set of all functions \( \nu : L \rightarrow M^* \) such that \( \nu(l) \in \mu (l)^* \).

By \( \mu (l)^* \) we denote the set of finite sequences over \( \mu (l) \). Mathematically, the set \( V_T \) is the Cartesian product of the sets \( \mu (l)^* \), for each \( l \). Effectively, transaction vectors are \( n \)-tuples of sequences where each coordinate corresponds to a leaf in the transaction tree (hence, \( n \) is the number of leaves) and contains a finite sequence of actions that have occurred on that leaf or access point.

When an action occurs on a leaf of the transaction tree, that is to say when an action associated with some subtransaction takes place on an access point, it appears on a new transaction vector and at the appropriate coordinate. For example, the vector \((s_1, \Lambda, \Lambda)\) describes that portion of behaviour of the transaction in which an action \( s_1 \) (e.g. service invocation) has taken place on the corresponding service allocated to the first coordinate. We use \( \Lambda \) to denote the empty sequence.

The vector \((s_1, s_2, \Lambda)\) describes that portion of behaviour in which both \( s_1 \) and \( s_2 \) have happened on the corresponding services while the vector \((s_1, s_3, s_2, \Lambda)\) describes an occurrence of \( s_1 \) and an occurrence of \( s_3 \) on the service corresponding to the first coordinate, and an occurrence of \( s_2 \) on the second coordinate. Nothing has happened on the service corresponding to the third coordinate.

In can be seen that each transaction vector provides a snapshot of behaviour in which the transaction has executed the actions appearing on the vector’s coordinates – it describes what actions have already occurred and on which part of the transaction tree. This vector-based description of behaviour allows to record the actions of a transaction as these occur on the multiple services involved in its execution. Readers familiar with process algebras like CSP or CCS can understand each particular coordinate of the vector description as a sequential CSP process. In this sense, the transaction vectors can be understood as the Cartesian product of sequential processes describing each leaf in a transaction tree.
Operations on vectors. Let $u, v \in V_T$ be transaction vectors, we define
- $u \leq v$ to be the unique vector $w$ such that $w(l) = u(l)v(l)$, for each $l \in L$ (concatenation)
- $u \leq v$ iff $u(l) \leq v(l)$, for each $l \in L$ (prefix ordering)
- $\operatorname{glb}(u,v)$ to be the vector $w$ such that $w(l) = \min\{u(l),v(l)\}$, for each $l \in L$
- $\operatorname{lub}(u,v)$ (if it exists) to be the vector $w$ such that $w(l) = \max\{u(l),v(l)\}$, for each $l \in L$
- if $u \leq v$, then we define $v / u$ to be the unique element $z \in V_T$ such that $u \cdot z = v$ (right-cancellation)

Thus, the operation of concatenation on vectors is defined in terms of the concatenation of sequences appearing on their respective coordinates. For example,

$$ (s_1s_3, s_2, \Lambda)(\Lambda, s_4, \Lambda) = (s_1s_3s_2s_4\Lambda) $$

The ordering amongst vectors is defined in terms of the usual prefix ordering operation on sequences appearing on their coordinates. For example, $(s_1, s_2, \Lambda) \leq (s_1s_3, s_2, \Lambda)$ since $s_1 \leq s_1s_3$ and $s_2 \leq s_2$ and $\Lambda \leq \Lambda$. In other words, the second vector ‘wins’ on the first coordinate (since it has a sequence of greater length in this coordinate) while the two vectors draw on all other coordinates.

It is not hard to see that some vectors will be incomparable. It turns out that such vectors describe either parallel or alternative behaviours of the transaction in question, and this will be further discussed in the following sections.

The operations $\operatorname{glb}()$ and $\operatorname{lub}()$ give the greatest lower bound and the least upper bound, respectively of $u, v \in V_T$, in the usual sense of lattices and domain theory [11]. As we will see, these operations are central to the treatment of concurrency in our approach.

The right-cancellation operator ‘/’ says that if $u$ is a transaction vector describing an initial part of the behaviour described by $v$ so that $u \leq v$, then $v / u$ is the ‘continuation’ of $u$ that extends it to $v$. This operation is central to the treatment of compensations in our approach. We return to this discussion in the concluding section of the paper.

It is important to stress the fact that all operations on vectors are performed coordinate-wise and this simplifies the proofs but also makes the formal model feasible for implementation.

Transaction vectors can be seen to be built up from the empty vector $\Lambda$ by a series of concatenations with column vectors that represent actions. In fact, in describing the behaviour of a transaction we are interested only in those vectors describing (orderings of) actions that we expect the transaction to engage in during the course of its execution.
This is the subset of all possible transaction vectors, over a given \( T \), we referred to as transaction language.

### A. Sequential service compositions

The prefix ordering relation on transaction vectors can be viewed as an ordering on partial executions, where each vector corresponds to that portion of behaviour in which the transaction has already engaged in the actions appearing on its coordinates. This can be expressed more succinctly by saying that \( u \leq v \) in a transaction language means that \( u \) is an earlier part of behaviour leading to \( v \).

A more careful examination of the mathematical construction shows that we can say more than that. Indeed, we find it useful to determine immediate predecessors (or successors) of a transaction vector.

**Cover.** Suppose that \( u, v \in V \subseteq V_T \). We say that \( v \) covers \( u \) in \( V \), and we write \( u \triangleleft v \), if

(i) \( u \leq v \) and \( u \neq v \) and

(ii) If \( z \in V \) such that \( u \leq z \leq v \), then \( z = u \lor z = v \).

Thus, whenever \( u \leq v \), and we also have that \( u \triangleleft v \), then the last actions that went into forming each vector have occurred in sequence - one after the other. This allows to model sequential dependency among services inside a transaction. Recall the example of Fig. 2 where service \( s_I \) feeds service \( s_S \).

Fig. 5 presents the corresponding schema for sequential service composition. The ‘TransactionVectors-Type’ has a TransactionVector element which includes a single transaction vector (e.g. describing the service invocation \( s_I \)) and the immediate successor (describing the service invocation \( s_S \)). The latter is captured by the ‘FollowedBy’ element. The type of ‘SequentialComposition’ simply includes a single transaction vector followed by another transaction vector.

### B. Parallel service compositions

Our approach towards modelling concurrent actions, actions that can happen in parallel, draws upon the concepts in Shields’ vector languages \([10]\) and Mazurkiewicz trace languages \([12]\) where concurrent events are considered as being unordered, in contrast to CSP trace theory where it is assumed that observations are sequential in nature and concurrent events are understood to occur in either order (non-deterministic interleaving).

The treatment of concurrency within our formal model of transactions thus takes up on non-interleaving models of concurrency, which introduce additional structure into formal languages in order to describe non-sequential behaviour. The additional structure is given in terms of an independence relation over action symbols, which describes potential concurrency.

In fact, the independence relation is a binary relation, defined over a set of actions, that is symmetric (to reflect the fact concurrency is always mutual) and irreflexive (to prohibit considering an action as being concurrent with itself). Intuitively, the independence relation \( \triangleleft \) defined over a set of actions \( A \) gives rise to an equivalence relation on sequences formed over \( A \). Now we have seen that transaction vectors are essentially tuples of sequences, hence we may make use of this construction in terms of the sequences formed over the sets of actions \( \mu(l) \), for each \( l \in L \), of a transaction.

In terms of our notation it is appropriate to say that the independence relation on the set of actions \( A \) of a transaction equates all, and only those, sequences over \( \mu(l) \), for each \( l \in L \), which differ in the order of adjacent and independent actions. Note that when the independence relation is empty in the sets \( \mu(l) \), for each \( l \in L \), no actions can be concurrent in the corresponding sequences \( \mu(l)^* \), for each \( l \in L \), which amounts to our understanding of sequential transaction processing systems (e.g. as described by processes in CSP and its extension with compensations in \([15]\)).

Drawing upon the extension of the independence relation \( \triangleleft \) to behaviour vectors in \([10]\), the notion of independence between actions in Mazurkiewicz traces can be readily interpreted into transaction vectors in our approach.

**Independence.** For \( u, v \in V \subseteq V_T \), we define

\[
\text{ind} \; u \; \leftrightarrow \; \forall l \in L: \; \mu(l) > \Lambda \; \Rightarrow \; \nu(l) = \Lambda
\]

This definition says that two transaction vectors are independent if the behaviours they describe concern distinct services (correspond to activation on different leaves of the corresponding transaction tree). This means that the behaviours described by \( u \) and \( v \) may occur independently.

In the case of column vectors, independence captures the fact that actions appearing in one vector may occur independently of those appearing in the other. If in addition
In discussing concurrent actions in a long-running transaction, we saw that the two incomparable transaction vectors represent concurrent behaviour. The vector they both cover is in fact their greatest lower bound and is obtained by applying the operation \( \text{glb}(t) \) given earlier. The fact the two incomparable vectors represent concurrent actions is only because they are bounded above in the set (by the transaction vector which is their lub) and is sitting on top of the lozenge). Whenever this latter requirement does not hold we may talk about events in conflict.

It might be instructive to make the distinction in terms of pictures and associated Hasse diagrams. In the diagram of Fig. 7, \( s_1 \) and \( s_2 \) are sequential (\( s_2 \) can only be invoked after \( s_1 \)) in Fig. 7(i) while there is a choice between them (alternative) in Fig. 7(ii), and they are concurrent in Fig. 7(iii).

Notice that the set of vectors in (i) does not include \((\Lambda, s_2, \Lambda)\), which means that \( s_2 \) never occurs before \( s_1 \); in (ii) it does not include \((s_1, s_2, \Lambda)\) which means there is no valid behaviour of the transaction processing system in which both \( s_1 \) and \( s_2 \) have taken place; in (iii) it includes all four vectors, which means that \( s_1, s_2, s_1 \) and both \( s_1 \) and \( s_2 \) are all valid observations of the behaviour of the transaction system in which \( s_1 \) and \( s_2 \) happened concurrently. This is indicated by the familiar lozenge shape that exhibits the characteristic structure of a finite lattice.

In further explanation, the vector sitting at the top of the lozenge is the \( \text{glb}(t) \) of the two incomparable vectors \((s_1, \Lambda, \Lambda)\) and \((\Lambda, s_2, \Lambda)\) sitting at the middle of the lozenge while the vector at the bottom is their \( \text{lub}(t) \). The lozenge as a whole describes that part of behaviour of the transaction in which \( s_1 \) and \( s_2 \) happened concurrently, as indicated by the vector at its bottom. Further details on how the ordering relations between actions are manifested in the resulting order structure of the resulting set of vectors can be found in [13].

Fig. 8 presents the alternative service composition schema. An ‘AlternativeComposition’ consists of two or more (in fact, up to the number of its access points; here 5) alternative transaction vectors. Notice that in this case there is no ‘FollowedBy’ element as vectors describing alternative service deployment do not lead to a common vector, i.e. do not have a \( \text{lub}(t) \) in the formal language.

Therefore, in our approach given the tree structure of a transaction we may derive a formal description of its intended behaviour, in terms of activations of its subtransactions and the coordination between them. The resulting behaviour patterns (see Fig. 7) can be analysed before
run-time as a means of preventing certain anomalies (such as race conditions) which could result in unexpected behaviour when the transaction actually takes place [13].

V. IMPLEMENTATION

We have used JAX-WS 2.1, Java 5, and Netbeans IDE 5.5.1 in our implementation framework. The JAX-WS is a new standard for message passing, it supports both synchronous and asynchronous message passing by the polling model and the call-back model, respectively. In the polling model, the client continuously polls the service response. In the call-back model, the client creates a call-back handler. JAX-WS is shown to perform better than JAX-RPC in certain aspects [14]. Furthermore, JAX-WS supports static and dynamic generation of web service client stubs.

In our implementation, the JAX-WS call-back message passing and dynamic client stub generation are used. The call-back message passing is suitable for asynchronous message passing, which is important for long-running transactions. The dynamic WS client creation enables web service invocation chains. Three participants are required in order to make a service X work: 1) X web service; 2) X TransactionAgent web service; 3) the client stubs for the X web service. The dependencies between web services are captured in an XML file, as discussed before.

From the transaction vector schema, we can create different transaction scenarios (as an XML file). The generated XML file shows the various states of a transaction for a specific scenario. We have designed a software agent coordinator which can perform the coordination of the service invocations as prescribed by the transaction vector language in a fully distributed manner. As a case study we analyse a simple transaction, with a sequential service composition, which involves the interactions shown in the sequence diagram of Fig. 9.

Our simple transaction scenario involves two participants, located at two different SMEs (Hotel and Taxi services). The initiator agent has an AgentHelper that communicates with the TransactionAgents. The transaction involves four players: HotelTransactionAgent (HTA; local coordinator of the Hotel), HotelService (webservice of the hotel), TaxiTransactionAgent (TTA; local coordinator of taxi), and TaxiService (webservice of the taxi). The AgentHelper plays the Initiator role and it starts the transaction by sending messages (setting the transaction context) to both HTA and TTA. This specifies the first web service should be deployed on the first access point (HTA) and the second will be deployed only after receiving the successful confirmation of the first (sequential service composition).

The following information about the required distributed coordination can be derived: 1) the AgentHelper will know that after it initiates the HTA and TTA, it needs to wait for a response from the TTA for the final preparation status; 2) the HTA will know that it depends on TTA so it needs to send a message to TTA asking for final preparation status; 3) the TTA will know that it gets a request message from the HTA and then sends its response message back to the AgentHelper for the final preparation status.

Fig. 10 shows the performance analysis of the case study in terms of CPU performance, memory usage, and thread usage when the client performs a commit transaction. The Netbeans 5.5.1 source is available following [9].

VI. CONCLUSIONS AND FUTURE WORK

We have described an integrated view of various aspects of a transaction model for digital ecosystems for business. What makes our approach particularly suitable for digital ecosystems involving SMEs is that it does not require a central point of control or knowledge of the local state of execution of the different partners involved in a long-running business transaction.

To harness the complexity in this highly dynamic environment and increase confidence in a successful outcome, a thorough understanding of the underlying service executions and their dependencies is essential. This has been given by a formal model that can be represented by standard XML schemas and be used to guide the implementation.

We laid the foundations for the formal modelling of long-running transactions in terms of the underlying service executions which are orchestrated locally. This allows to identify the necessary behavioural patterns in terms of the (orderings of the) service invocations, and can be exploited in ensuring that compensating actions, whenever necessary, are also performed in the right order. The right-cancellation operator $\land'$, when applied to a vector and its cover (immediate predecessor), isolates the last action that occurred in that portion of behaviour. This is precisely the action that needs to be compensated at that given point.

The true-concurrent formal semantics allows internal activities of a transaction to communicate and can capture
parallelism. Unlike other approaches that use process algebras to model long-running transactions such as [15],[16], in our approach there is no need to consider different sequences of actions within a transaction and then compose them in order to model concurrency. Concurrency is handled in terms of the actions themselves, and there is no need for all actions within a transaction to be independent (and isolated). This is one of the benefits of opting for a non-interleaving semantics, based on [10], [12], as it allows modelling true-concurrency between actions in a long-running transaction.

An extension to address partial results (intermediate results released outside a transaction), based on [17], and compensating actions is currently under investigation at both the modelling and implementation levels. Preliminary results [18] are encouraging.

The current implementation framework is independent of the underlying network supporting the transactions. Early analysis indicates that the design of the supporting network architecture and the transaction model are interrelated. Aspects of a P2P network design targeted at increasing resistance to failure and fragmentation can enhance the recovery mechanisms applied at the transaction level, and vice versa. Work is in progress on integrating the implementation of the transaction model with the P2P network described [18].

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

VII. REFERENCES