Open Philosophies for Associative Autopoietic Digital Ecosystems

Contract n° 034824

Workpackage 3: Autopoietic P2P networks

Deliverable D3.2: Report on formal analysis of autopoietic P2P network, together with predictions of performance

Project funded by the European Community under the “Information Society Technology” Programme

Contract Number: 034824

Project Acronym: OPAALS

Title: Open Philosophies for Associative Autopoietic Digital Ecosystems

Deliverable N°: D3.2

Due dates: 06/2007

Delivery Date: 08/2007

Short Description:

This document is the second in the series of deliverables from WP3 which is focused on the development of a fully distributed P2P architecture to support open and trusted collaborations between SMEs to ensure their sustainability within a pan-European Digital Ecosystem. The first deliverable detailed the OPAALS distributed transaction model and identified the demands it makes on the supporting P2P network, and set out the preliminary architecture required. This second deliverable focuses on the formal analysis of both the transaction model and the characteristics of the underlying P2P network architecture. In particular, it introduces a formal model for describing the behaviour of the underlying service executions involved in a long-running transaction, a design that allows their coordination in a fully distributed manner, and an analysis of performance for the most critical parts of the supporting P2P network infrastructure.

Author: UniS

Partners contributed: TechIDEAS

Made available to: Public

VERSIONING		
VERSION	DATE	AUTHOR, ORGANISATION
0.1	**06/2007**	**AMIR RAZAVI, SOTIRIS MOSCHOYIANNIS, PAUL KRAUSE**
0.9	**07/2007**	**AMIR RAZAVI, SOTIRIS MOSCHOYIANNIS, PAUL KRAUSE**
1.0	**07/2007**	**AMIR RAZAVI, SOTIRIS MOSCHOYIANNIS, PAUL KRAUSE**
1.1	**08/2007**	**AMIR RAZAVI, SOTIRIS MOSCHOYIANNIS, PAUL KRAUSE**

Quality check:

1st Internal Reviewer : Thomas Kurz (SUAS)

2nd Internal Reviewer: Paul Malone (WIT)

Table of Contents

1. Introduction

The development of sustainable digital ecosystems involves a number of different views which can be seen to merge in the need for a thorough understanding of the continuous creation and sharing of knowledge in the form of business models, software infrastructure for open e-business transactions, and new formal and semi-formal languages. From a computer science point of view the realisation of such an ecosystem involves the development of the core components for an Open Knowledge Space and the development of the formal foundations for the autopoiesis and evolution in peer-to-peer (P2P) networks and collaborative services needed to realise distributed long-lived transactions in open communities of small-to-medium enterprises (SMEs).

We should emphasise that whilst this report focuses on business-to-business interactions, both the work on the transaction model and that on the P2P network are also targeted at open knowledge sources. We view the absence of a transactional model in the asynchronous-messaging Ajax world, for example, as a serious weakness that limits the applicability of this technology to the Open Knowledge Space (OKS).

The main objective of Workpackage 3 has been to investigate the most effective P2P architecture and dynamic topology that can support distributed long-running transactions and the automatic aggregation of their underlying services in digital ecosystems. The complexity of business-to-business (B2B) interactions in such a highly dynamic environment entails the need for the development of a distributed transaction model that overcomes certain limitations in existing transactional environments as well as a formal foundation for the distributed orchestration of the underlying services. The P2P architecture is intended to enable open and trusted collaborations between SMEs and thus our interest in B2B relationships rather than business-to-consumer (B2C) relationships, which are often more easily constrained and controlled.

Drawing upon the understanding of the mechanisms at work in living organisms, which engage in a large number of complex and distributed interactions, the metaphors of evolution and autopoiesis seem to provide the most natural starting point for the conception of truly distributed P2P architecture to support open e-business transactions in a way that allows no critical dependencies on single organisations, and no critical points of failure. The resulting *autopoietic* P2P architecture should provide a self-maintaining environment that evolves to adapt to the complexity emerging from the large number of interactions between entities that are organised recursively in smaller and simpler networks.

In the first deliverable from Workpackage 3, deliverable D3.1 [RMK07], we provided an extended analysis of a digital ecosystem for business along with details of the OPAALS distributed transaction model. This led to the specification of the main requirements for the supporting P2P network to support distributed long-running transactions, and a preliminary design of a truly distributed P2P network that would support open transactions within a community of SMEs. This second deliverable from WP3 provides a more detailed formal analysis to both components of a Digital Ecosystem, namely the distributed transaction model and the supporting P2P network.

Business transactions in a 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. In this report we describe a formal model of long-running multi-service transactions and the distributed orchestration of the underlying service executions. The formalisation of the interaction-based composition is aimed at describing the behavioural patterns services should follow in order to guarantee successful commitment or compensation within the transaction model.

The execution of a long-running transaction corresponds to conducting a business activity whose parts are offered as web services. These services need to be coordinated properly to increase confidence in a successful outcome. Therefore, each party engaging in a long-running transaction must handle not only the

application-specific aspects of the various service executions but also their interactions with one another, in terms of the ordering of service invocations. The challenge is that such coordination of the underlying service executions needs to be performed in a fully distributed manner. This is a prerequisite for realising the benefits of the Service-Oriented Computing (SOC) paradigm but is also in line with the demands for local autonomy of SMEs. It is further reinforced by the nature of the supporting P2P architecture which is based on the premise of openness or inclusiveness of the run-time environment as well as the absence of vulnerability to single points of failure. To address this challenge we need a carefully refined design of the structure of each Local Coordinator in setting up and executing a long-running transaction.

The network infrastructure intended to facilitate B2B transactions in a digital ecosystem needs to take into account a variety of factors coming from the nature of long-running multi-service transactions. The availability of services, but also of nodes themselves, as well as issues of unpredictable disconnection are of primary concern, as identified in deliverable D3.1 [RMK07]. In addition, there are further aspects evident in 'scale free' networks such as fragmentation that remain to be addressed. Further, a digital ecosystem involving open communities of SMEs is a highly dynamic environment and thus it is important to consider the way the network responds to changes in the number of nodes. We discuss an evolutionary framework, inspired by biology, and describe how it can be adapted and applied to the P2P network considered in WP3 in a way that respects its primary characteristics.

Whilst this report focuses on the formal modelling work, it should be emphasised that work is currently underway to implement both the transaction model and the P2P architecture. This will be reported on in later deliverables.

The report is structured as follows. Chapter 2 provides a formal semantics of long-running transactions using a vector-language based description of behaviour. This approach has a number of advantages over alternative models. These are also discussed in this chapter. Chapter 3 then describes the local agent structure that is needed to support distributed long-term transactions across a P2P network. Chapter 4 discusses the results of the first phase of our performance analysis. This analysis needs to be extended with further experimentation, which is under way. We conclude the report in Chapter 5 with some pointers to next steps.

2. Distributed Transactions in Digital Ecosystems

In this chapter we are concerned with modelling long-running transactions in digital ecosystems. The aim is to provide a formal foundation for the coordination of the underlying service executions which can provide a thorough understanding of the behaviour patterns that these service compositions should exhibit in order to guarantee that either the transaction commits or is compensated for.

It is only recently that long-running transactions, as understood in a business environment rather than in traditional database systems, have received the attention of the formal methods community. After a brief description of the basic characteristics of long-running transactions and the surrounding issues in modelling their critical aspects, we outline formal approaches to modelling transactions. The discussion focuses on which transactional aspects are covered and which are missing. We then introduce a formal model for long-running multi-service transactions in digital ecosystems for business and show how it can be used for a behavioural analysis in terms of the underlying service compositions and the compensation mechanism that needs to be deployed if some failure later in the transaction makes it necessary.

2.1 Long-running Transactions: Features and Characteristics

Within the database community the conventional definition of a transaction [Dat96] is based on ACID properties: **A**tomicity – either all tasks in a transaction are performed, or none of them are; **C**onsistency – data is in a consistent state when the transaction begins, and when it ends; **I**solation – all operations in a transaction are isolated from operations outside the transaction; **D**urability – upon successful completion, the result of the transaction will persist. However, in advanced distributed applications these properties can present an unacceptable limitation and reduce performance dramatically, a view also supported in [Elm94].

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 [CCC+04b, RKM06]) 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.

This makes the adoption of the Service-Oriented Computing (SOC) [Pap03] more relevant than ever. 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. This means that a service must be designed in a way that it can be invoked by various service clients and is logically decoupled from any service caller. The actual architectural approach of SOC is called SOA [PaG03, PTD+06] and is particularly applicable when multiple distributed applications are running on varied technologies and platforms need to communicate with each other. This would allow enterprises to mix and match services to perform business transactions in a digital ecosystem with the minimal programming effort.

Although our primary concern in WP3 of OPAALS is to provide support for long-running transactions in digital ecosystems involving open communities of SMEs, the basic characteristics of long-running transactions pertain to the more general interactions within a digital ecosystem, and this is mostly due to the fully distributed nature of the environment. Such characteristics have been analysed in Deliverable D3.1 [RMK07] and are only briefly outlined in the sequel. 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, Atomicity and Isolation are questionable.

Business transactions typically involve interactions and coordination between multiple partners. A long-running transaction comprises a number of *sub-transactions* or *activities* which involve the execution of a number of underlying services. It is often the case that internal activities 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 the service invocations or due to the sharing of data. Booking a flight ticket before booking a hotel room at the destination is an example for the former, while booking a taxi or buying a bus ticket, depending on the remaining budget, is an example of the latter.

Further, in a digital ecosystem for business a number of distributed long-running multi-service transactions are expected to take place. Many business scenarios require that a transaction releases some results to another transaction, before it commits. For example, consider a organising the travel for a meeting, which includes booking a flight, booking a hotel, and then either go for booking a taxi or buying a ticket for the bus, depending on the remaining budget. (Note that this already indicates the need for exchange of results within a transaction.) Now if there is a really good deal for all the requirements (possible in terms of distance between hotel and meeting place, etc.), but it would require that the attendee goes slightly overspent, the ‘travel booking’ transaction (before finalising the bookings/purchases) may contact another transaction, the 'budget handling' transaction say, to seek approval for a slightly increased budget.

In other words, dependencies may exist across transactions due to the need for releasing *partial results* outside a transaction. This may not happen as often as releasing data inside a transaction but is nevertheless a central requirement for distributed long-running transactions if they are to cover a wide range of business models. Conventional transaction models such as Sagas [G-MS87] or the more recent models *Web Services Transactions* (WS-Tx) [CCJ+04],[CCC+03] and *Business Transaction Protocol* (BTP) [FDF+04] do not provide capability for partial results and inevitably make it the business process designer’s responsibility. This often means that an ad-hoc complicated transaction needs to be designed or, even worse, results in adding new transactions that do not reflect the exact needs of the business model itself, but rather are incorporated to get round the problem. It can be seen that designing transactions with full ACID properties limits significantly the coverage of business requirements and is thus not suitable for e-business in general, and digital ecosystems in particular.

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 *autopoietic* P2P network in OPAALS, whose preliminary design was described in D3.1 [RMK07] and is formally analysed in later parts of this report, together with the proposed fully distributed transaction model go some way to alleviate these concerns.

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 which 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) [CCJ+04],[CCC+03] and *Business Transaction Protocol* (BTP) [FDF+04], as pointed out in Deliverable D3.1 [RMK07] in which a more comprehensive discussion can be found.

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. Another serious side-effect has to do with triggering compensations and updates in the corresponding accountability and trust frameworks sitting on top of the transaction layer. (These aspects are considered in WP4 of OPAALS.) 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 transpires that 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.

Transaction models that have been designed with web services in mind, and are currently widely used in practice, include WS-Tx [CCJ+04], [CCC+03], [CCC+04a], [CCC+04b] and BTP [FDF+04], which have been reviewed in deliverable D3.1 [RMK07]. 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.

Distributed long-running multi-service transactions involve the execution of the underlying service compositions which need to be coordinated (service orchestration) in a principled manner. Harnessing the complexity of the service composition and the call interplay between services is a necessary first step for increasing the confidence in a successful outcome in a transactional environment. In other words, if we are to provide a P2P architecture and dynamic topology that can support distributed long-running transactions in open communities of SMEs, and eventually the automatic aggregation of services in digital ecosystems, we need to provide a formal foundation for long–running transactions and the distributed orchestration of the underlying service executions.

In what follows we review some very recent approaches to the formalisation of long-running transactions and in light of these approaches we then describe work in progress on the formal modelling and analysis of the transaction model developed in OPAALS.

2.2 Formal Approaches to Modelling Transactions

Formal approaches to transaction modelling have been introduced recently, and some will be discussed in greater details in the following sections, aiming to give a formal semantics to long-running transactions and in that way address some of the surrounding issues. A common denominator seems to be the attempt to relax ACID properties by organising a long-running transaction into a series of activities. The idea is that

each activity is a discrete transactional unit of work that can either commit or abort (in which case the previously committed activities need to be undone or rolled back).

2.2.1 A process algebra for transactional aspects in composed services

In this section we provide a brief outline of a formal approach to modelling the primitives of long-running transactions with a version of a process algebra that draws upon the well-known process algebras *Calculus for Communicating Systems* (CCS) [Mil80] and *Communicating Sequential Processes* (CSP) [Hoa85]. Our discussion is based on the work presented in [BMM05] which epitomises this process algebraic approach to defining a formal language for describing the complex interactions between the services involved in the execution of long-running transactions.

In particular, the authors in [BMM05] set out to define a set of primitives for long-running transactions in flow composition languages concerned with structured control flows. In other words, flows that can be defined in terms of primitives like sequencing and branching (choice). This gives a formal semantics to long-running transactions, whose flow of control exhibits these features, and the authors show that the semantics is adequate for the modelled features. In what follows we give a brief outline of this work and discuss it in view of the issues surrounding long-running transactions in digital ecosystems.

The approach to modelling transactions described in [BMM05] is driven by the understanding of transactions as in *Sagas* [G-MS87]. The *Sagas* model is one of the points of reference for long-running or long-lived transactions, nevertheless it has been criticised for its limited applicability in conducting open e-business long-running transactions [FuG05]. That said, *Sagas* was one of the first models for long-running transactions and some of the key concepts are still relevant in any transaction model. The key idea is that valid executions of a transactional business process (or of a part of it) are those that complete all involved activities. This is essentially the Atomicity part of ACID properties.

Since business activities are long-running in nature, as we have discussed in the previous section in more detail, a more relaxed version of atomicity can be achieved by associating processes with compensating activities that can recover or 'undo' the effects of previously completed activities within a transaction (partial executions). The question of course is how the compensating mechanism can be applied to ensure consistency of data at all times and take into account the dependencies that may exist both inside and outside of a transaction. Therefore, our discussion of [BMM05] focuses on this aspect in the remaining of this section rather than the inherent limitations of the *Sagas* transaction model (they have chosen to adopt) itself.

We start by presenting the formalisation of a subset of *Sagas*, namely the sequential composition (of activities or actions inside a saga) and then the parallel composition of different sagas.

A *sequential saga*, as mentioned in [G-MS87] and used in [BMM05] to model a long-running transaction, is a sequence of atomic *activities* (called *sub-transactions* or *actions* or *steps*) that should be executed completely. The parallel execution of several sagas can interleave actions in any way, but any single action is considered to be atomic. The interleaving of actions, a concept found in CSP and CCS, means that one action is executed at a time and the ordering of execution respects the sequencing of the actions in each saga. Actions are atomic in the sense that either they are successfully executed (committed) or no effect is observed when the execution fails (aborted). In other words, actions inside a saga adhere to ACID properties. Additionally, every action Ai in a saga has a compensating activity Bi that can be executed to 'undo' the effects of a successful execution of Ai upon a later failure.

Any partial execution of a saga is undesirable – it must either execute all its sequential actions successfully or not at all – if an action fails during the execution of a saga then the whole saga must be compensated for. Formally, a saga involving $A_1, \ldots A_n$ (where each A_i is associated with a compensating

action B_i) is guaranteed to execute either the sequence A_1; … ; A_n (successful execution of the whole saga) or the compensated sequence A_1; … ; A_j ; B_j; …. B_1 for some $j < n$ (in case the action A_j fails, execution stops and the compensating actions are executed in the reverse order).

The formalisation of sequential sagas in [BMM05] comprises a syntax for writing sequential sagas and a semantics that determines the outcome of a saga and how their compensations are executed in the event of a failure. Let A be an infinite set of names for actions (denoted by A, B, …) and let $0 \notin A$ denote a special action that always completes and has no effect. The syntax for sequential sagas considered in [BMM05] is given in Figure 2.1.

X ::= 0 | A | A ÷ B	(STEP)
P :: = X | P ; P	(PROCESS)
S ::= {[P]}	(SAGA)

Fig. 2.1 Syntax of sequential sagas in [BMM05]

It can be seen that a sequential saga (or a long-running transaction comprising sequential actions or sub-transactions only) consists in a sequential process P. Each action in P is either an action A or a compensated action A ÷ B, where B is the compensating action of A. The ';' operator applied on processes means that process on the left is composed sequentially with the process on the right. The term 0 represents a process (modelling a long-running transaction) that does nothing and always completes successfully. It is difficult to imagine what long-running transaction would be modelled by 0, but such a construct seems to be necessary to the proposed formalism. We return to this discussion in the sequel.

In defining the semantics of long-running transactions (as sagas) constructs that capture the outcome of the execution and the valuation of the constituent actions in each step during execution (referred to as *context*) are required.

The approach of [BMM05] considers the case that the compensating actions inside a saga (the B_j; … ; B_1 part of the execution of a saga) may also fail. Hence, there are three different outcomes captured in a set R = {■, ◙, ◘} where ■ represents successful execution (*commit*), ◙ represents failure with successful compensation (*compensated abort*) and ◘ represents failure with unsuccessful compensation (*abnormal abort*). Also, let □ range over R.

A context is required to determine whether actions were completed successfully or not. This is denoted by Γ and is defined in [BMM05] as a partial function over A that maps any activity to the result obtained during its execution, i.e. $\Gamma : A \mapsto \{■, ◙\}$. Actions can only commit or do compensated abort, they do not terminate abnormally (only processes do). Hence, Γ provides and evaluation on each action inside a saga and the semantics of a sequential saga are given by the relation

$$\Gamma \vdash S \xrightarrow{\alpha} \square$$

which is used to denote that the execution of saga S produces □ when the actions behave like Γ. The observation α describes the flow of control occurring when executing S under Γ. The flow α is also a sequence of actions, but its actions have no compensating activities. More details can be found in [BMM05]. The semantics of this relation, and consequently of sequential sagas, is given Figure 2.2, as found in [BMM05].

$$\Gamma \text{ |-- } \langle 0,\beta\rangle \xrightarrow{0} \langle \blacksquare , \beta'\rangle \qquad \text{(ZERO)}$$

$$A \mapsto \blacksquare ,\ \Gamma \text{ |-- } \langle A \div B,\beta\rangle \xrightarrow{A} \langle \blacksquare , B;\beta'\rangle \qquad \text{(S-ACT)}$$

$$\frac{\Gamma \text{ |-- } \langle \beta, 0\rangle \xrightarrow{\alpha} \langle \blacksquare, 0\rangle}{A \mapsto \blacksquare, ,\ \Gamma \text{ |-- } \langle A \div B,\beta\rangle \xrightarrow{\alpha} \langle \blacksquare , 0\rangle} \qquad \text{(S-CMP)}$$

$$\frac{\Gamma \text{ |-- } \langle \beta, 0\rangle \xrightarrow{\alpha} \langle \blacksquare, 0\rangle}{A \mapsto \blacksquare, ,\ \Gamma \text{ |-- } \langle A \div B,\beta\rangle \xrightarrow{\alpha} \langle \square , 0\rangle} \qquad \text{(F-CMP)}$$

$$\frac{\Gamma \text{ |-- } \langle P, \beta\rangle \xrightarrow{\alpha} \langle \blacksquare, \beta''\rangle \quad \Gamma \text{ |-- } \langle Q, \beta''\rangle \xrightarrow{\alpha'} \langle \square, \beta'\rangle}{\Gamma \text{ |-- } \langle P;Q , \beta\rangle \xrightarrow{\alpha;\alpha'} \langle \square, \beta'\rangle} \qquad \text{(S-STEP)}$$

$$\frac{\Gamma \text{ |-- } \langle P, \beta\rangle \xrightarrow{\alpha} \langle \sigma, 0\rangle \quad \sigma \in \{\blacksquare, \square\}}{\Gamma \text{ |-- } \langle P;Q , \beta\rangle \xrightarrow{\alpha} \langle \sigma, 0\rangle} \qquad \text{(A-STEP)}$$

$$\frac{\Gamma \text{ |-- } \langle P, 0\rangle \xrightarrow{\alpha} \langle \square, \beta\rangle}{\Gamma \text{ |-- } \{[P]\} \xrightarrow{\alpha} \square} \qquad \text{(SAGA)}$$

Figure 2.2 Semantics of sequential sagas in [BMM05]

The auxiliary relation Γ |-- $\langle P,\beta\rangle \xrightarrow{\alpha} \langle \square, \beta'\rangle$ describes the behaviour of a process P within a saga that has already installed compensation β (β represents a process without compensations just like α). The overall idea is that when P is executed it either commits or aborts or fails, but additionally it can change the compensations, for instance by installing new compensations β', as in rule (S-ACT).

Rule (ZERO) states that 0 always commits without changing the installed compensations. Rule (S-ACT) describes the successful execution of the compensated action A÷B when A commits. Sequential composition is essentially handled in rules (S-CMP) and (F-CMP) which describe the execution of A÷B when A fails in a saga that has already installed compensations β. In particular, the rule (S-CMP) states the case in which the compensation procedure completes successfully. In rule (F-CMP) compensation procedure fails, in which case the process terminates abnormally (indicated by '□'). Rule (S-STEP) handles the behaviour of a process P;Q when P commits, in which case execution continues with Q taking into account the compensations produced by (the execution of) P. Rule (A-STEP) describes the behaviour of P;Q when P terminates abnormally, in which case the compensation is activated as soon as P aborts.

Next, we present the ideas in [BMM05] concerning the concurrent execution of actions. This is handled by introducing an operator '|' denoting the parallel composition of processes. Parallel composition is

understood as in the CSP [Hoa85] where sequential processes can be executed in parallel so long as they synchronise (both are ready to engage in that action) on common actions while all non-common actions from each are interleaved (that is, they can occur in either order so long as the sequencing in the process they come from is preserved).

It is important to note that in the approach of [BMM05] to modelling long-running transactions, processes that are composed in parallel do not communicate. In other words, they do not have common actions on which they have to synchronise. This means that their respective actions are all independent and hence composition simply results in all possible interleavings of the actions from each. A similar approach to parallel composition of sequential processes is taken in *compensating* CSP [BHF05] which is discussed in greater detail in the next section (Section 2.2.2).

The syntax for parallel sagas is basically that of sequential sagas extended with the '|' operator, as given in Figure 2.3.

X ::= 0 | A | A ÷ B	(STEP)
P :: = X | P ; P | P | P	(PROCESS)
S ::= {[P]}	(SAGA)

Figure 2.3 Syntax of parallel sagas in [BMM05]

It is required that '|' is associative and commutative with identity 0 (the nil term). Also, sequential composition is considered to have higher binding precedence than parallel composition.

As for sequential sagas, a parallel saga terminates successfully only when all its actions commit. The whole saga (all participating sequential processes in the parallel saga) will be compensated when an action is aborted. The compensations for parallel sagas are executed concurrently (instead of in the reverse order as in sequential composition).

The approach considers initially a semantics for parallel sagas where the sequential processes that comprise a parallel saga are completely independent. This has the disadvantage that if one sequential process aborts, then there is no way to immediately enforce the rest of the sequential processes to abort too. This can only be done once the rest of the processes have completed their execution.

To alleviate this problem the authors in [BMM05] subsequently introduce two new types of results for a process within a saga: ◙ denoting that the process has been forced to compensate, and it has been compensated successfully, and ◘ denoting that the process has been forced to compensate but the compensation itself has failed.

Let σ_1, σ_2 range over $\underline{R} = R \cup \{◙, ◘\}$. The result obtained by combining the execution of two parallel processes is given in the table of Figure 2.4, as found in [BMM05].

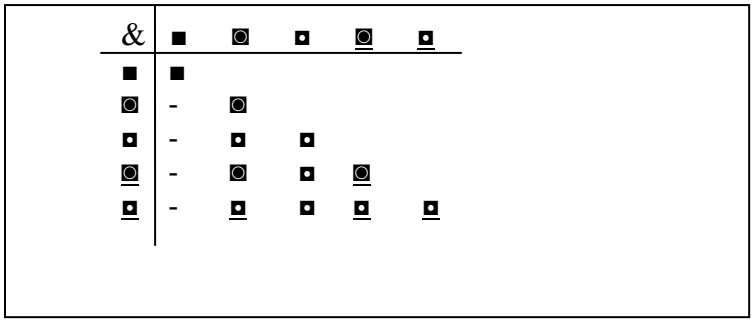


Figure 2.4 Combining the result of sequential processes in parallel sagas [BMM05]

The '&' operator that combines the result of sequential processes in a parallel saga is associative and commutative. This means the upper half of the table is not required and is thus empty. More details can be found in [BMM05].

The semantics of parallel sagas, is given Figure 2.5, as found in [BMM05]. All rules of sequential sagas (given earlier in Figure 2.2) remain unchanged, but the rule (A-STEP) now also considers the new results ◙, ◘ for forced compensation and four new rules are added.

Γ |-- <0,β> $\xrightarrow{0}$ <■ , β'> (ZERO)

$A \mapsto$ ■ , Γ |-- <A÷B,β> $\xrightarrow{A}$ <■ , B;β'> (S-ACT)

$$\frac{\Gamma \text{ |-- } \langle \beta, 0\rangle \xrightarrow{\alpha} \langle \blacksquare, 0\rangle}{A \mapsto \boxed{\blacksquare}, \ , \ \Gamma \text{ |-- } \langle A \div B, \beta\rangle \xrightarrow{\alpha} \langle \boxed{\blacksquare} , 0\rangle} \quad \text{(S-CMP)}$$

$$\frac{\Gamma \text{ |-- } \langle \beta, 0\rangle \xrightarrow{\alpha} \langle \boxed{\blacksquare}, 0\rangle}{A \mapsto \boxed{\blacksquare}, \ , \ \Gamma \text{ |-- } \langle A \div B, \beta\rangle \xrightarrow{\alpha} \langle \boxed{\square} , 0\rangle} \quad \text{(F-CMP)}$$

$$\frac{\Gamma \text{ |-- } \langle P, \beta\rangle \xrightarrow{\alpha} \langle \blacksquare, \beta''\rangle \quad \Gamma \text{ |-- } \langle Q, \beta''\rangle \xrightarrow{\alpha'} \langle \square, \beta'\rangle}{\Gamma \text{ |-- } \langle P;Q \ , \beta\rangle \xrightarrow{\alpha;\alpha'} \langle \square, \beta'\rangle} \quad \text{(S-STEP)}$$

$$\frac{\Gamma \text{ |-- } \langle P, \beta\rangle \xrightarrow{\alpha} \langle \sigma, 0\rangle \qquad \sigma \in \{\boxed{\blacksquare}, \boxed{\square}, \underline{\boxed{\blacksquare}}, \underline{\boxed{\square}}\}}{\Gamma \text{ |-- } \langle P;Q \ , \beta\rangle \xrightarrow{\alpha} \langle \sigma, 0\rangle} \quad \text{(A-STEP)}$$

$$\frac{\Gamma \text{ |-- } \langle P, 0\rangle \xrightarrow{\alpha} \langle \blacksquare, \beta'\rangle \quad \Gamma \text{ |-- } \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \blacksquare, \beta''\rangle}{\Gamma \text{ |-- } \langle P \mid Q \ , \beta\rangle \xrightarrow{\alpha|\alpha'} \langle \blacksquare \ , \beta' \mid \beta'' \ ; \beta\rangle} \quad \text{(S-PAR)}$$

$$\frac{\Gamma \text{ |-- } \langle P, 0\rangle \xrightarrow{\alpha} \langle \sigma_1, 0\rangle \quad \Gamma \text{ |-- } \langle Q, 0\rangle \xrightarrow{\alpha} \langle \sigma_2, 0\rangle}{\Gamma \text{ |-- } \langle P \mid Q \ , \beta\rangle \xrightarrow{\alpha|\alpha'} \langle \sigma_1 \& \sigma_2, 0\rangle \qquad \sigma_1 \in \{\boxed{\square}, \underline{\boxed{\square}}\}, \ \sigma_2 \in \{\boxed{\blacksquare}, \boxed{\square}, \underline{\boxed{\blacksquare}}, \underline{\boxed{\square}}\}} \quad \text{(F-PAR)}$$

$$\frac{\Gamma \text{ |-- } \langle P, 0\rangle \xrightarrow{\alpha} \langle \sigma_1, 0\rangle \quad \Gamma \text{ |-- } \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \sigma_2, 0\rangle \quad \Gamma \text{ |-- } \langle \beta, 0\rangle \xrightarrow{\gamma} \langle \square_1, 0\rangle}{\Gamma \text{ |-- } \langle P \mid Q \ , \beta\rangle \xrightarrow{(\alpha|\alpha')'\gamma} \langle \sigma_1 \& \sigma_2 \& \square_2 \ , 0\rangle \qquad \sigma_1, \sigma_2 \in \{\boxed{\blacksquare}, \underline{\boxed{\blacksquare}}\} \text{ and } \square_2 = \boxed{\blacksquare}, \text{ if } \square_1 = \blacksquare, \text{ and } \underline{\boxed{\square}}, \text{ otherwise}} \quad \text{(C-PAR)}$$

$$\frac{\Gamma \text{ |-- } \langle \beta, 0\rangle \xrightarrow{\alpha} \langle \square_1, 0\rangle}{\Gamma \text{ |-- } \langle P, \beta\rangle \xrightarrow{\alpha} \langle \square_2, 0\rangle \qquad \square_2 = \underline{\boxed{\blacksquare}}, \text{ if } \square_1 = \blacksquare, \text{ and } \underline{\boxed{\square}}, \text{ otherwise}} \quad \text{(FORCED-ABT)}$$

Figure 2.5 Semantics of parallel sagas [BMM05]

The new rules (S-PAR), (F-PAR) and (C-PAR) specify the behaviour of parallel composition of sequential processes, covering all possible outcomes, i.e. commit, compensated abort and abnormal abort. The sequential processes run in parallel without initial compensations. If both processes commit (rule S-PAR), the original compensation β is updated with the compensations β' and β'' from each. In particular, if the whole parallel composition P | Q has to be compensated then β' and β'' are executed in parallel and β is started only when they have both finished.

If one process has started its compensation procedure then the other process must also be compensated. Rule (F-PAR) says that if one process fails during its compensation, then the final result for the parallel composition P | Q is a (possibly forced) abnormal termination, i.e. ◙ or ◘. In this case the compensation β installed initially is never executed. Rule (C-PAR) describes the case in which both P and Q are successfully compensated for. In this case the compensation β installed initially is also executed. The new rule (FORCED-ABT) handles the forced compensation of a process P, i.e. P can activate the compensation procedure before starting its execution that will produce a forced termination, i.e. ◙ or ◘.

In addition, there is a notion of nesting in the formal approach of [BMM05] for long-running transactions. This concerns nested transactions, and there is some level of nesting in sagas [G-MS87], where a long-running transaction is decomposed into a hierarchy of activities or sub-transactions. In this scheme, the root of the hierarchy is the *top-level transaction* and any sub-transaction executes independently and concurrently with respect to its parent and siblings, deciding autonomously whether to commit or abort. In short, the authors in [BMM05] give a construction that allows a process P (either in a sequential composition P ; Q or in a parallel composition P | Q) to abort and compensate successfully without needing to inform its parent in the nested saga. Thus, the saga can commit even if one of its sequential processes P has aborted, providing it has been successfully compensated for. If it does not compensate successfully, then this can no longer be hidden from the parent and the whole saga aborts.

The consideration of nested transactions and nested sagas is rather interesting but it is not entirely clear in the treatment of [BMM05] how this nesting capability is different to an alternative path of execution for a transaction (which would be captured using choice in the CSP-oriented formalism of the proposed approach). In other words, it is not clear whether nesting has been submerged with the non-deterministic choice found in process algebras such as CSP and CCS. This perhaps comes down to how the hierarchy is defined within the structure of a nested transaction. This is not covered in [BMM05] but if the examples in the paper can be used to infer the hierarchy, it seems to be the case that a nested sub-transaction corresponds to precisely an additional path of execution that allows the transaction to commit. If it aborts, then the alternative path (referred to as the sibling in [BMM05] is executed and the transaction can still commit.

Continuing with some overall comments on the process algebraic approach to modelling long-running transaction proposed in [BMM05], it seems appropriate to say that it is very closely tied to the well-known approaches of Communicating Sequential Processes [Hoa85] and, to a lesser extent to the, Calculus of Communicating Systems [Mil80]. This means it suffers from some of the limitations inherent in these approaches and in particular CSP. For example, the use of a nil process 0 is a necessary feature of CSP, but here sequential processes are used to model long-running transactions, and it is hard to see what transaction a business process designer would model using 0 (recall its meaning is that it is a process which always terminates successfully and has no effect).

Moreover, the fact that only sequential blocks of actions are considered, and these do not communicate whether they are composed in sequence or in parallel, means that the resulting formalism is not expressive enough to capture partial results (which would require communication between sequential processes) and is even limited in capturing the dependencies that arise inside a long-running transaction. The compensating mechanism used follows exactly that of the forward actions and there is no provision to capture omitted results or forward recovery (unless the nesting scheme is indeed a means of introducing alternative scenarios) and leaves little flexibility for extension with such features in the future.

In addition, there are certain problems that arise in compensating for parallel processes due to the fact that the compensation procedure should be independent of any particular interleaving of actions. This is not possible within CSP since the parallel composition operator is defined to do just that – interleave non-common actions in any way. The fact there is no communication between sequential processes that are composed in parallel in [BMM05] means that the parallel composition operator does no more than simply generate the non-deterministic interleavings of the actions from each process. In fact, the extension to CSP with compensations to produce the so-called *compensating CSP* [BHF05], paradoxically considers a non-interleaving semantics in performing the compensations for sequential processes that are composed in parallel. This approach is discussed next.

2.2.2 Extending CSP with compensations

This line of work is proposing a model of long-running transactions within the framework of the Communicating Sequential Processes (CSP) [Hoa85] process algebra. For this purpose, an extension of the CSP formalism with compensating actions has been considered in [BHF05], [RiB06]. The resulting *compensating CSP* has been used for modelling long-running transactions, which are understood as a sequence of isolated activities. This notion of a long-running transaction draws upon the concept found in *Sagas* [G-MS87]. A saga partitions a long-running transaction into a sequence of several smaller activities or sub-transactions, where each has an associated compensation. If one of the sub-transactions in the sequence fails or aborts, the compensations associated with the previously committed sub-transactions is executed in the reverse order.

In this section we briefly outline this approach to modelling long-running transactions focusing on how the internal activities and their compensations can be orchestrated in *compensating CSP*.

The behaviour of a process (denoted by P, Q, …) can be recorded as a sequence or trace in CSP terminology (denoted by p, q, …) of all its environmentally observable actions (denoted by A, B, …). The CSP formalism makes use of some special internal actions, like √ indicating successful termination of a process. In compensating CSP, additional special actions are needed such as !, indicating an interrupt throw, and ?, indicating an interrupt yield. In fact, compensating CSP also requires some special processes like THROW in addition to the standard CSP sequential processes. The purpose of these shall become more clear in the sequel.

As in [BHF05] we will use <A, B, √> to denote the trace that describes the behaviour of a process P = A;B which executes action A, then action B and then terminates successfully. Processes can be composed to produce composite processes, such as sequential composition (P;Q) or parallel composition (P||Q). The traces of composite processes are defined in terms of the traces of their constituent processes. A formal semantics will be given below. Before that we summarise the syntax used in standard processes of CSP in Figure 2.6.

P, Q	:: = A	(atomic action)
	| P ; Q	(sequential composition)
	| P □ Q	(choice)
	| P || Q	(parallel composition)
	| SKIP	(normal termination)
	| THROW	(throw an interrupt)
	| YIELD	(yield to an interrupt)
	| P►Q	(interrupt handler)
	| [PP]	(transaction block)

Fig. 2.6 Standard processes in CSP

The presentation of the semantics for the standard processes that follows is based on that given in [BHF05], [RiB06] but it should be noted that it follows the standard trace semantics of the usual Communicating Sequential Processes of [Hoa85].

Assume a process has an alphabet of actions Σ and let $\Omega = \{\surd, !, ?\}$ be the set of special actions of processes (not included in Σ) which are also referred to as *terminal events* in [BHF05]. The concatenation of traces s and t is denoted by st. Standard processes are defined as non-empty sets of traces of the form $s<\omega>$ where $s \in \Sigma^*$ and $\omega \in \Omega$. Thus,

- $s<\surd>$ is a trace leading to successful termination
- $s<!>$ is a trace leading to interrupt throw
- $s<?>$ is a trace leading to interrupt yield

The process that performs a single atomic event and terminates successfully consists of a single complete trace. This essentially is used to denote an atomic action.

Atomic action. For $A \in \Sigma$, $A = \{<A, \surd>\}$

A process P is a sequence or trace of actions from the corresponding alphabet Σ. The sequential composition of processes P;Q is done in a way that execution of the sequence of actions of process Q commences when that of P has completed successfully. Thus, successful traces of P are followed by traces of Q while other traces of P are not. This is summarised in the following definition given in [BHF05].

Sequential composition. $p<\surd> ; q = pq$
$p< \omega > ; q = p< \omega >$, if $\omega \neq \surd$

In terms of processes, the ';' operator is defined as $P ; Q = \{p ; q \mid p \in P \wedge q \in Q\}$

The choice between two processes is defined as the union of their traces, as in usual CSP. To the best of our understanding this is a correct adaptation of the definition found in [BHF05] (given here in terms of the trace semantics rather than in terms of processes themselves, as done originally in [BHF05]).

Choice. $P \square Q = \{ p \cup q \mid p \in P \wedge q \in Q\}$

In the Communicating Sequential Processes [Hoa85], the parallel composition of two processes is defined in a way that 1) the processes synchronise on common actions (one waits for the action to be enabled in the other process too) and 2) the processes asynchronously execute actions that are not common to both.

In the compensating CSP framework for long-running transactions proposed in [BHF05], processes that are composed in parallel are not allowed to communicate. In other words, the synchronous execution of observable actions is not supported. Hence, point i) of the definition of parallel composition of processes is not applicable in compensating CSP.

The occurrences of the non-common actions from separate processes (point ii) above) are modelled by considering their non-deterministic interleaving. For example, for two actions A and B executed in parallel, their parallel composition A||B means that A can be followed by B or B can be followed by A. In other words, the two actions are perceived as occurring in either order. Note that this interpretation is different to that of non-interleaving approaches which consider concurrent actions as occurring in any order.

Interleaving. $p \,|||\, \langle\rangle = \{p\}$
$\langle\rangle \,|||\, p = \{q\}$

$\langle x\rangle p \,|||\, \langle y\rangle q = \{\langle x\rangle r \mid r \in (p \,|||\, \langle y\rangle q\} \cup \{\}\langle y\rangle r \mid r \in (\langle x\rangle p \,|||\, q)\}$

In the compensating CSP framework for long-running transactions proposed in [BHF05], processes that are composed in parallel are not allowed to communicate. In other words, the synchronous execution of observable actions is not supported. Hence, point i) of the definition of parallel composition of processes is not applicable in compensating CSP. Processes that are composed sequentially (using the ';' operator described above) do not communicate anyway, so the proposed formalism does not support any form of communication or interaction between activities inside a transaction. Since transactions are modelled by processes, which do not communicate, there is also no communication across transactions. Thus, the proposed framework does not support the release of results inside or outside a transaction.

In compensating Communicating Sequential Processes, processes composed in parallel synchronise only on successful termination (the '√' of each) and on interruption ('!' and '?' from each). If ω and ω' are terminal events from each of the processes P and Q, then the joint terminal event ω&ω' of the parallel composition P||Q is determined by the table given in Fig 2.7.

ω	ω	ω&ω'
!	!	!
!	?	!
!	√	!
?	?	?
?	√	?
√	√	√

Fig. 2.7 Synchronisation on terminal events in compensating CSP [BHF05]

The parallel composition of processes is defined (in terms of their traces) to be the set of interleavings of their observable part followed by the synchronisation of their terminal events.

Parallel composition. $p\langle\omega\rangle \parallel q\langle\omega'\rangle = \{r\langle\omega\&\omega'\rangle \mid r \in (p \,|||\, q)\}$

In terms of processes , the '||' operator is defined as $P \parallel Q = \{r \mid r \in (p \,|||\, q) \wedge p \in P \wedge q \in Q\}$

The '||' is commutative and associative (although this is not shown in [BHF05] or [RiB06]). Further, [BHF05] give the following law that governs the interaction between '||' and an interrupt.

$$\text{THROW} \parallel (\text{YIELD};P) = \text{THROW} \,\square\, P;\ \text{THROW}$$

A careful reading of the law shows that interrupt does not have priority over other events. The authors in [BHF05] claim this is to be expected since it is not desirable that the entire system responds immediately to an attempt by one party to raise an interrupt. However, as we are to read failure for interrupt and consider a digital ecosystem for business as the distributed setting, it is not only desirable but essential for all parties involved in a business activity to know immediately about failure at some part of the system – this would allow to localise the affected part and trigger the recovery mechanism before the failure propagates and results in costly chains of rollbacks or, even worse, brings the whole system to a an abrupt halt.

In what follows we present the additional special constructs, actions and operators, included in extending Communicating Sequential Processes with compensations in order to model long-running transactions.

There is a special process, termed SKIP, which does nothing and always terminates successfully.

Skip. SKIP = {<√>}

There is another special process defined in [BHF05], termed THROW, which does nothing (notice there is no action from the alphabet involved) and always fails or aborts. This is captured in the proposed formalism by the fact it generates a unique trace which contains the special action '!' denoting an interrupt. It is later used in compensating CSP to indicate the failure of an activity (or sub-transaction). This special process is defined as follows.

Throw. THROW = {<!>}

There is yet another special process defined in [BHF05], termed YIELD, which is used in compensating CSP to indicate that a process is aware of an interrupt thrown by one of its actions. This process does nothing and stops (yields), denoted by '?' or does nothing and successfully terminates, denoted by '√'.

Yield. YIELD = {<?>, <√>}

This process generates two kinds of traces; either yields, denoted by '?', or successfully terminates, denoted by '√'. This means that the composite process P ; YIELD ; Q either does P, and then does '?' (yields), or does P and then does '√' (successfully terminates after P). Hence, Q is never executed.

The intention behind these constructs is to extend the usual formalism of Communicating Sequential Processes with a recovery and compensation mechanism for a transactional environment. The mechanism boils down to being able to signify failure, and then alert the rest of the transaction, so that the required compensations are executed. Hence, the special action '!' is used to describe the failure of a transaction. The intended effect is that '!' causes an immediate disruption of the flow of control. A process always yields to an interrupt '!', whether it is ready to terminate (indicated by √), and no additional symbol or special action is used, or it is in the middle of its execution, in which case it again yields and this is indicated by '?'.

A special operator, termed interrupt operator and denoted by '►', is given in [BHF05], to define the handling of interrupts. The operator is applied on traces and it says that if a '!' occurs in the trace on the left, then execution continues with the trace on the right; if it does not occur, then the trace on the right (presumably this is the 'compensation trace') is never executed. The operator is defined as follows.

Interrupt handler. $p\langle ! \rangle \blacktriangleright q = pq$
$p\langle \omega \rangle \blacktriangleright q = p\langle \omega \rangle$, if $\omega \neq !$

In terms of processes, the '►' operator is defined as $P \blacktriangleright Q = \{ p \blacktriangleright q \mid p \in P \wedge q \in Q \}$

In a sequential process, the effect of '►' is understood to be that the current flow of execution stops and the compensation actions (one for each successfully executed action) are executed in the reverse order. In parallel processes (or, to be more precise, in processes which have been composed in parallel using the '||'

operator), the effect of an interrupt '!' in one process is that the '►' causes the rest of the processes to disrupt their flow of control and yield to the interrupt. In other words, the whole group of parallel composed processes have to execute their compensating actions or fail (if the compensating actions are not executed successfully).

We may now proceed to describe how the compensations take place. That is to say, how the compensating CSP framework goes about performing the compensations, in the reverse order to that of the forward actions, to undo the effects of the transactions up to the point of failure. First, we describe the notion of *compensable* process, given in [BHF05] and then outline the cancellation semantics of such processes that is designed to allow for the rollback of a transaction.

A compensable process is understood as a process that contains forward behaviour and compensation behaviour. The compensation is to be executed to compensate for the forward action, if necessary, i.e. if some failure occurs before the transaction commits. The compensation part of a compensable process is also a trace comprising *compensating* actions. For each action A, the programmer is supposed to provide a compensating action, denoted by A°, whose occurrence after A will restore the system to a state which is an acceptable approximation of the state that it had before the start of the transaction. Note that the phrase 'an acceptable approximation of' rather than 'the same' state is used here, as in some cases (e.g. when a human user is involved) some actions of a transaction cannot be really undone.

In this way, the primitive component of a long-running transaction is written as $A \div A^{\circ}$, where A is the forward action and A° is the corresponding compensating action. A process P can also be declared to have its own compensation, for example $P \div Q$, in which case the compensation Q overrides the compensations declared inside P (those that would accrue from its forward actions). The syntax for compensable processes (denoted by PP, QQ, …) is given in Figure 2.8.

PP, QQ	:: = P ÷ Q	(compensation pair)
	| PP ; QQ	(sequential composition)
	| PP □ QQ	(choice)
	| PP || QQ	(parallel composition)
	| SKIPP	(normal termination)
	| THROWW	(throw an interrupt)
	| YIELDD	(yield to an interrupt)

Figure 2.8 Compensable processes in compensating CSP

The behaviour of a compensable process in the proposed model for transactions is captured by a pair of traces of the form $(p\langle\omega\rangle, p'\langle\omega'\rangle)$ where $p\langle\omega\rangle$ is a forward trace and $p'\langle\omega'\rangle$ is the corresponding compensation trace. In what follows we present the semantics of compensable processes given in [BHF05].

The choice of compensable processes is the same as in standard processes.

Choice. $PP \square QQ = PP \cup QQ$

The parallel composition of compensable processes is also as in standard processes. To the best of our understanding, this presents some issues when it comes to actually performing these compensations. As we will see, the authors attempt to alleviate such problems by appealing to a non-interleaving semantics. Remember that the parallel composition of processes in CSP is done in an interleaving manner, and hence

this also applies to compensable processes. We will have more to say about this when we present the cancellation semantics for compensating CSP in the sequel.

Compensable parallel composition. $(p, p') \parallel (q, q') = \{(r, r') \mid r \in (p \parallel q) \wedge r' \in (p' \parallel q')\}$

In terms of compensable processes, the '||' operator is defined as

$$PP \parallel QQ = \{rr \mid rr \in (pp \parallel qq) \wedge pp \in PP \wedge qq \in QQ\}$$

The sequential composition of compensable processes is defined so that the composition behaviour of the first process is made to happen after that of the second process. Behaviours of a compensable process PP which lead to failure, i.e. where the forward trace $p<\omega>$ is unsuccessful remain unchanged in the sense that the compensation trace is executed but the second process does not get executed at all (neither the forward nor the compensation part).

Compensable sequential composition. $(p<\surd>,p') ; (q, q') = (pq, q';p')$
$(p<\omega>, p') ; (q, q') = (p<\omega>, p')$, if $\omega \neq \surd$

In terms of compensable processes, the ';' operator is given as

$$PP ; QQ = \{pp ; qq \mid pp \in PP \wedge qq \in QQ\}$$

Now a compensable process can be defined as a construction from two standard processes as follows. In the pair $P \div Q$, the successful forward behaviour from P is augmented with compensating behaviour from Q resulting in a compensable process. If P throws an interrupt or yields to an interrupt (i.e. fails) then the compensation part is empty.

Compensation pair. $p<\surd> \div q = (p<\surd>, q)$
$p<\omega> \div q = (p<\omega>, p')$, if $\omega \neq \surd$

In terms of compensable process the pair is given by

$$P \div Q = \{(<?>, <\surd>)\} \cup \{p \div q \mid p \in P \wedge q \in Q\}$$

Notice that when applied to processes the definition contains an additional behaviour, which is intended to allow a compensation pair to yield immediately with the empty compensation.

The authors in [HF05] also give compensable versions of the special processes introduced in compensating CSP.

Compensable special processes.

SKIPP = SKIP ÷ SKIP
THROWW = THROW ÷ SKIP
YIELDD = YIELD ÷ SKIP

The construct of transaction block is used to bring all actions necessary for compensating sequential processes in CSP under the same umbrella.

Transaction block.

$$[PP] = \{pp' \mid (p<!>, p') \} \cup \{p<\surd> \mid (p<\surd>,p') \in PP\}$$

The concept of a transaction block involves running the compensation part of interrupted forward traces, discarding the compensation parts of the terminating forward traces and completely removing traces whose forward parts are yielding. There are certain conditions required to ensure that a transaction block [PP] is

not empty. (Note that non-emptiness of PP is not sufficient to ensure non-emptiness of [PP].) The conditions given in [BHF05] declare that all processes, standard and compensable, consist of some terminating or interrupting behaviour. In other words, either '√' or '!' must at least be included in the corresponding traces.

The compensating Communicating Sequential Processes of [BHF05], [RiB06] introduce a cancellation semantics for compensable processes in order to examine how the effect of forward actions is cancelled by compensating actions. Central to this cancellation theory is a notion of *independence* between actions. The use of an independence relation is central to well-known *non-interleaving* models for concurrency [Shi85, Shi97, Maz88] which do not identify concurrent execution with the non-deterministic interleaving of actions. We return to the discussion on the sudden adoption of a non-interleaving semantics by compensating CSP below. First, we describe its cancellation semantics for transactions, as given in [BHF05].

Let F be a set of forwards actions and C be a set of compensating actions, and $F \cap C = \varnothing$. The authors define a relation *cancel* on F and C so that *cancel*(A, A°) means that A° cancels the effect of A. Next, the authors define a cancellation function *C* on traces, which removes action-compensating action pairs, like A and A°, from the traces of a compensable process. Naturally, it is desirable that the coverage of the cancellation function is increased as much as possible, i.e. it captures more pairs of actions and removes them. This is where the need for defining an independence relation arises.

Independence. The independence relation is never defined formally in [BHF05] so we cannot give a formal definition as we did for the other constructs used in compensating CSP. The authors declare that certain actions are independent in the sense that they can occur in *either* order. This, they claim, would typically be the case for compensations of parallel processes. They write *independent*(A,B) to indicate that A and B may be transposed in a trace as they do not interfere with each other. Further, the authors assume that *independent* is symmetric. It is not clear how such an assumption can be made when independent has not been defined as a relation. We return to this and other issues regarding the use of an independence relation in compensating CSP in the sequel, and after we complete the presentation of the cancellation semantics.

The cancellation function on traces is defined as follows. (Notice it makes use of the undefined *independent* relation described above.)

Cancellation function. If a trace t is of the form $p<A>q<A^{\circ}>r$, and if *cancel*(A, A°) and for each $B \in q$ we have independent(A°,B) then:

$$C(<A>q<A^{\circ}>r) = C(pqr)$$

If the trace t does not satisfy the above conditions then no further cancellation can be applied and hence in this case we have,

$$C(t) = t$$

Thus, the cancellation function in compensating Communicating Sequential Processes [BHF05] is intended to remove matching pairs of action-compensating action from the resulting trace of the transaction or compensable process. Further, we have seen that a compensable process may (and typically will) have more than one trace. This may be due to the choice operator or due to the parallel composition operator which generates all possible interleavings of the actions from each process. This is precisely where the independence relation comes in. It can be used to filter out different traces that represent concurrent execution of the same actions, as done in non-interleaving approaches to concurrency.

We note that the use of an independence relation on an alphabet of actions is central to well-known *non-interleaving* models for concurrency [Shi85, Maz88, Shi97] – in fact, independence is used in this models to describe potential concurrency. This is not mentioned in [BHF05] but the essence of the independence relation in compensating Communicating Sequential Processes (CSP) is precisely the notion of independence used in non-interleaving models for true-concurrency. A comprehensive survey of different concurrency models can be found in [WiN95].

The set up of the cancellation semantics for transactions in compensating CSP aims at the characterisation of what the authors refer to as *self-cancelling* compensable processes. These are processes whose behaviour is described by traces which after application of the cancellation function result in the empty trace. This means that whatever was done during the execution of the transaction has been effectively undone. We see no reason for the introduction of the new term as this seems to be precisely the purpose of defining compensation mechanism and recovery management in transaction models.

The example given in [BHF05] confirms this rationale. Consider the trace $\langle A,B,C, C^{\circ}, A^{\circ}, B^{\circ}\rangle$ for which we have that *cancel*(A, A°) and *cancel*(B, B°) and *cancel*(C, C°) and also that A° and B° are independent. Successive applications of the cancellation function to the trace gives:

$C(\langle A,B,C, C^{\circ}, A^{\circ}, B^{\circ}\rangle) = \langle A,B, A^{\circ}, B^{\circ}\rangle$ since *cancel*(C, C°)

$C(\langle A,B, A^{\circ}, B^{\circ}\rangle) = \langle A, A^{\circ}\rangle$ since *independent*$(A^{\circ}, B^{\circ},)$ and *cancel*(B, B°)

$C(A, A^{\circ}) = \langle\rangle$ since *cancel*(A, A°)

It can be seen that this example uses the fact that because A° and B° are independent they can be transposed in the corresponding trace. Recall a trace is a sequence. However, this is not the case in mathematics. The fact that two action symbols are related by independence in the alphabet does not infer that they can be transposed in any sequence.

What the theory underlying the non-interleaving models, in which an independence relation on the alphabet of actions is central, tells us is that an independence relation on an alphabet needs to be defined as a symmetric and irreflexive binary relation. In fact, as pointed out in [Mos05] symmetry requires that concurrency is always mutual and irreflexivity prohibits an action being concurrent with itself. This relation then needs to be lifted onto sequences formed over the given alphabet, where adjacent independent actions are allowed to permute, and the resulting sequences or traces are related by 'independence'. Then by taking the reflexive, transitive closure of this second 'independence' relation on traces, we get an equivalence relation on traces, which is what is used to model concurrency in non-interleaving approaches.

We will see a proper definition of an independence relation in Section 2.3, when we describe a formal approach to modelling long-running transactions in which concurrency is handled in a non-interleaving manner. This in fact draws upon established theories of vector languages [Shi79] and asynchronous transition systems [Shi85] and Mazurkiewicz traces [Maz88]. The overall idea is that independent actions which appear consecutively in a sequence of execution are concurrent.

Returning to the compensating CSP and the appeal to a non-interleaving semantics in examining the relation between forward actions and their compensations, it can be seen that the intention is to be able to recover more traces than adhering to the interleaving semantics laid out up to that point. This is only possible if there is a notion of equivalence between traces – the traces before and after cancellation must be equivalent. This is not the case with the current set up in compensating CSP.

The appeal to a non-interleaving semantics is encouraging as it would allow for a wider spectrum of traces, a wider range of long-running transactions modelled using these traces, to be compensated for. However, this must be set up properly, and there is existing literature on non-interleaving models of concurrency [Shi85, Maz88] which have been making use of the independence relation for more than 25 years. Additionally, even if the transition to a non-interleaving semantics when it comes to compensating a transaction is done properly, the benefit of starting with an interleaving semantics is not clear.

Moreover, the trace of the example given in [BHF05] and repeated in this report, cannot result in any other way following the semantics of compensable processes in compensating CSP but only as an interleaving of parallel processes. Its compensation can only be done by applying a non-interleaving

semantics. This is not pointed out in [BHF05], which is evidence that reinforces the argument of whether the interleaving semantics is the most suitable option to start with.

It can also be argued that the special processes like SKIP or THROW or YIELD are not intuitive when it comes to modelling long-running transactions. They are an important part of the proposed framework as it is important for example in CSP, and also in compensating CSP, to capture the successful termination of a process. This is the purpose of the special process SKIP used in [BHF05], RiB06], and it is even more critical to the parallel composition of processes since these synchronise on their terminating actions. However, processes in compensating CSP are used to describe long-running transactions and hence, it is difficult to see what transaction would be modelled by SKIP as all this process says is that it is a transaction that does nothing and always terminates successfully. A similar argument holds for THROW, which is a process that does nothing and always fails; difficult to imagine a business process designer thinking of transactions that do nothing and always succeed or always fail. So these are constructs that are proprietary to the specific formalism used, that of compensating CSP, and have little to do with the modelling domain itself.

On a related note, sequential and parallel execution of transactions is rather peculiar in the framework of compensating CSP. Recall that CSP [Hoa85] stands for Communicating Sequential Processes and thus the primitive constructs in this case are processes, which in turn comprise sequences of actions. Hence, the trace semantics we have seen here. This means that sequential and parallel execution of actions (but also of transactions at the top level) cannot be expressed in a straightforward manner. It is defined on sequences of actions (processes) and not on actions themselves. To define concurrent actions within a transaction (something rather common in B2B scenarios) one has to consider a process that comprises a single action. This is in contrast to the overall concept of thinking of a transaction as a sequential process. What compensating CSP can do naturally is to consider the sequential and parallel composition of different sequences of actions. The main benefit of the CSP approach however, has been the fact that it can put different processes in parallel by enforcing synchronisation on their common events (communication) and interleaving all their non-common events. This is the concept of parallel composition in the Communicating Sequential Processes of [Hoa85] (hence, the name). In compensating CSP however, processes are not allowed to communicate. There is no interaction whether they are composed in parallel or sequentially. This means it does not actually build on the main benefit of the CSP formalism. All it does is to take two sequences of actions and either compose them sequentially (execution of the second sequence commences when the first has terminated successfully - SKIP) or compose them in parallel (execution of the two sequences commences together, one action at a time, from either of the processes, in a way that respects the sequencing of actions in each – interleaving).

As a result of prohibiting any communication, the compensating Communicating Sequential Processes of [BHF05, [RiB06], have no provision for partial results but also, there is little evidence and little flexibility for capturing more than the dependencies due to ordering, i.e. due to data sharing between actions or sub-transactions. Note that there is no communication in either case – sequential or parallel composition – and thus there is no sharing of data or exchange of results inside of a transaction before it commits. This also raises the question of the relevance of a independence relation in the framework – all actions are independent by default: in sequence, they execute in isolation; in parallel, they are independent since there are no common actions and hence, no communication.

The only communication that is allowed is that of synchronising on terminal events of sequential processes that are composed in parallel. This however does not provide any leverage for covering partial results. Instead it prohibits the triggering of the compensating procedure in one process as soon as a failure is detected in some other process. In other words, it does not allow to enforce the abortion of one branch as soon as failure occurs in the other branch. This is not remotely satisfactory when modelling real problems which require activities within a transaction to be executed in parallel, since it may result in a situation where one processes fails relatively early in its execution and the other process (or, even worse, processes) have to complete their execution until they are ready to commit in order to be notified (via synchronisation on terminal events) that some process has failed. The situation becomes even worse when considering that these processes will have to run their compensation for the whole set of actions they executed while the

other process had already failed. It might be worth noting that this problem has been partly touched upon in the approach of [BMM05] discussed in the previous section, when the authors talk about a 'naïve semantics' to parallel composition (which had to be extended with a 'forced to compensate' notion.)

Finally, there is little evidence in the proposed compensating Communicating Sequential Processes of [BHF05], [RiB06] that a wide range of business scenarios involving long-running transactions can be faithfully modelled using sequences of actions that can be either composed in sequence (even though they already comprise a sequence themselves) or in parallel (but without communication between the parallel sequences). This set up is particularly tailored to follow the Communicating Sequential Processes of [Hoa85] but its applicability for long-running business transactions is questionable.

To paint the overall picture of this line of work, we add a final note on StAC (Structured Activity Compensation) [BCF+02], [BF04], a language in the spirit of CSP with exception handling mechanisms, developed by some of the same authors, namely Butler and Ferreira, as a tool for compensating CSP. The objective was to develop a language that, follows CSP, and considers compensations closely related to the control flow of the executed process. However, the execution of the compensations is not part of the definition of a transaction, but rather StAC has special primitives for activating the installed compensations. Consequently, compensations are not actually related to (or triggered by) the failure or success of the activities of processes, as usually expected in flow composition languages, e.g. BPEL4WS [BPE]. Moreover, as pointed out in [BMM05], there is a tight relation between data structures used by activities and the control flow of processes in StAc. This means that reasoning about processes in StAC requires low level description of activities which is in contrast with preserving the local autonomy of participating business' platforms. Furthermore, StAC comprises a large set of operators, more than what we have seen in compensating CSP [BHF05]. The operational semantics of these operators is given in yet another intermediate language $StAC_i$ [BF04], in which some usual behaviours of compensations are only achieved through a – sometimes obscure – combination of several $StAC_i$ operators. Since operators in StAC can only be understood by analysing their encodings in $StAC_i$ it is difficult to reason about the interplay between compensations, nesting (if any) and parallel composition in StAC. The promise that such issues would be rectified in compensating CSP [BHF05] has been, at best, moderately achieved – the improvement concerns the reasoning about the effects of a transaction in a compositional way, and in this way extend the semantics of StAC to make it compositional, while the aforementioned issues remain to be addressed.

2.2.3 A logic-based formalisation for service-oriented interactions

In this section we are more concerned with the coordination aspects of the services underlying the execution of a long-running transaction. The previous approaches have abstracted away from the underlying service executions required to perform a transaction, and have only considered the more general notion of an *action* inside a transaction and an associated compensating action that can be executed to 'undo' its effects. These actions however involve the execution of services on local platforms in a distributed setting such as a digital ecosystem and need to be coordinated effectively and in a distributed fashion in order to maintain openness and scalability.

As already mentioned before, it is the difficulty of effective coordination that has often led researchers and practitioners to adopt centralised approaches. In what follows we give an outline of a formal approach to coordinating distributed services that has been proposed in [Sin97] for constructing multi-agent systems out of autonomous agents, but the main ideas are highly relevant in a transactional environment.

In fact, the approach described in [Sin97] builds on techniques from workflow and relaxed transaction scheduling in databases. One of its overarching concerns is to provide support to the designer who must ensure the local *agents* or *platforms* (in which services reside) interact properly and in a way that respects

the local autonomy of the participants. It takes a view of coordination as a service that takes declarative specifications of the desired interactions and translates them into 'low-level' events which can then be scheduled through message passing among the local agents. The low level events correspond to significant, externally visible events that feature in the interactions.

The approach to coordination of services in distributed multi-agent systems taken in [Sin97] is based on temporal logic and includes a semantics of events and their use in message passing to implement control and data flow. The primary assumption is that the designer has limited knowledge of the local platform or the local agent's design. This knowledge is in terms of their externally visible actions, or *significant events*, which are significant for coordination purposes. Events in [Sin97] can be of the following four types:

- flexible, which the agent is willing to delay or omit
- inevitable, which the agent is willing to delay
- immediate, which the agent is neither willing to delay nor omit
- triggerable, which the agent is willing to perform based on external request

The significant events may be organised in a skeleton which can be represented as a finite state automaton that provides a simple model of the local agent's interactions. Rather than being an entity that lies beneath each platform, the coordination is distributed across the significant events of each platform or agent. The interactions using significant events are formalised in [Sin97] in an event-based linear temporal logic, I, which is a propositional logic augmented with a *before* '$\circ$' temporal operator. The literals in I denote event types, and can have parameters. A literal with all constant parameters denotes an event token. The *before* operator is formally a dual of the more familiar 'until' operator found in temporal logics.

Let Ξ denote all event literals (with constant or variable parameters), and let $\Gamma \subseteq \Xi$ contain only constant literals. A *dependency* is an expression in I and a workflow W is a set of dependencies. The syntax of I is given in Figure 2.9, as found in [Sin97].

- $\Xi \subseteq I$
- $I_1, I_2 \in I \Rightarrow I_1 \vee I_2,\ I_1 \wedge I_2,\ \ I_1 \circ I_2 \in I$

Figure 2.9 Syntax of coordination in [Sin97]

The formal semantics of the syntax for coordination is based on sequences of events, or traces in process algebraic terminology as we have seen. Central to the treatment is the notion of *consistent* traces, denoted by $\mathbf{U}_I$, which are those in which an event token and its complement do not occur, and in which event tokens are not repeated. $[[]] : I \rightarrow \wp(\mathbf{U}_I)$ gives the denotation of each member in I. The specifications in I select the acceptable traces – specifying I means that the service may accept any trace in $[[I]]$.

Constant parameters in [Sin97] are written as c_i, variables as v_i and either variety as p_i. Hence, $e[c_1 \ldots c_m]$ means that e occurs appropriately instantiated. This is stated in the first rule of the semantics in Figure 2.10. The complement of an event token e is denoted by $\underline{e}$. the intuition is that $\underline{e}$ is established only when it is definite that e will never occur. Complemented literals are included in Ξ and thus need no additional syntax or semantics rules. $I(v)$ refers to an expression free in variable v while $I(v::=c)$ refers to the expression obtained by substituting every occurrence of v by c. Complemented literals are quantified by the second rule of the semantics given in Figure 2.10.

1. $[[e[c_1 \ldots c_m]]] = \{\tau \in \mathbf{U}_I : e[c_1 \ldots c_m] \text{ occurs in } \tau\}$
2. $[[I(v)]] = \bigcap_{c \in C} [[I(v::=c)]]$
3. $[[I_1 \vee I_2]] = [[I_1]] \cup [[I_2]]$
4. $[[I_1 \wedge I_2]] = [[I_1]] \cap [[I_2]]$
5. $[[I_1 \circ I_2]] = \{ \tau_1\tau_2 \in \mathbf{U}_I : \tau_1 \in [[I_1]] \text{ and } \tau_2 \in [[I_2]]\}$

Figure 2.10 Semantics of coordination in [Sin97]

We have seen that preserving local autonomy is imperative in a digital ecosystem for business. This is a primary requirement in the logic-based approach to coordination of service interactions in [Sin97] which is one of the basic motivations for presenting this work here. In order to provide for distributed coordination, the events must take decisions based on local information. This requires determining the conditions, called *guards*, on the events by which decisions can be take upon their occurrence.

Guards are the weakest conditions that guarantee correctness if the associated event occurs. They are temporal expressions that say:
- which events have occurred (and which have not)
- which events have not occurred, but are expected to occur
- which events will never occur

T is the language in which guards are expressed and is expressive enough to capture the above distinctions between event occurrences (or not). It syntax is given in Figure 2.11, as found in [Sin97].

- $\Gamma \subseteq T$
- $E, F \in T \Rightarrow E \vee F, E \wedge F, E \circ F, \Box E, \Diamond E, \neg E \in T$

Figure 2.11 Syntax of specification language *T* in [Sin97]

In further explanation, $\Box E$ means that E will always hold, $\Diamond E$ means that E will eventually hold (and thus $\Box E$ entials $\Diamond E$), and $\neg E$ means that E does not (yet) hold. Also, $E \vee F$ means that either E or F has occurred, $E \wedge F$ menas that both E and F have occurred, and $E \circ F$ means that F has occurred preceded by E (i.e. both E and F have occurrred, but E has occurred before E).

The formal semantics for *T* is given in terms of a trace (as for *I* before) and an index into that trace. The index is used to characterise progress along a given computation and that is in turn used to determine the decision on each event. It is important to note that traces here are given in terms of sequences of events and not states. In addition, the semantics definitions (see Figure 2.12) are given in terms of a pair of indices rather than a single index (and hence [Sin97] is talking about intervals).

The expression $u \models_{i,k} E$, for $0 \leq i \leq k$, means that E is satisfied over the subsequence of the sequence u between i and k. The expression $u \models_k E$, for $k \geq 0$, states that E is satisfied on u at index k – implicitly, here i is set to 0. $\Lambda \triangleq$ denotes the empty trace. A trace u is *maximal* if and only if for each event, either the event or its complement occurs in u. Let $\mathbf{U}_I \triangleq$ denote the set of maximal traces. Finally, assume that $\Xi \neq \varnothing$, hence, also $\Gamma \neq \varnothing$. The formal semantics for the temporal logic language *T* are given in Figure 2.12, as found in [Sin97].

1. $u \models_i E$ iff $u \models_{0,i} E$
2. $u \models_{i\,k} f$ iff $(\exists j : i \leq j \leq k$ and $u_j = f)$, where $f \in \Gamma$
3. $u \models_{i\,k} E \vee F$ iff $u \models_{i\,k} E$ or $u \models_{i\,k} F$
4. $u \models_{i\,k} E \wedge F$ iff $u \models_{i\,k} E$ and $u \models_{i\,k} F$
5. $u \models_{i\,k} E \circ F$ iff $(\exists j : i \leq j \leq k$ and $u \models_{i\,j} E$ and $u \models_{j+1,\,k} F)$
6. $u \models_{i\,k} \perp$
7. $u \models_{i\,k} \neg E$ iff $u \not\models_{i\,k} E$
8. $u \models_{i\,k} \Box E$ iff $(\forall j: k \leq j \Rightarrow u \models_{i\,j} E)$
9. $u \models_{i\,k} \Diamond E$ iff $(\exists j: k \leq j \Rightarrow u \models_{i\,j} E)$

Figure 2.12 Semantics of language *T*, given in [Sin97]

The semantics rule 1, which involves just one index i, invokes the semantics with the entire trace until i. The second index is interpreted as the present moment in the computation. Rules 3, 4, 6 and 7 deal with the usual connectives found in temporal logic. Rules 8 and 9 involve looking into the future. Rule 2 and 5 capture the dependence of an expression on the immediate past, and this is bounded by the first index on the definition. Finally note that rule 5 introduces a non-zero first index.

For each computation the guards capture how far the specific computation ought to have progressed when the guarded event occurs, and what obligations would remain to realise that computation. To establish such reasoning, termed *residuation*, [Sin97] introduces an operator '/', defined by $/ : I \times \Xi \mapsto I$, which is not in *I* or *T*. Given a dependency D and an event e, the expression D/e gives the residual or 'what remains' of D after e occurs. The authors then give a set of equations which can be used to calculate the residual of D when an e occurs. The equations are given in Figure 2.13, where D is a sequence expression, and E is a sequence expression or $\perp$ (to our understanding the purpose of introducing $\perp$ is to allow the treatment of an atom as a sequence by using that $f \equiv f \circ \perp$). Γ_D denotes the set of actions occurring in D. Also, the denotation of any sequence $e_1 \circ \ldots \circ e_n$ in which (for $i \neq j$), $e_i = e_j$ or $e_i = \underline{e}_j$ is set to the empty sequence, i.e. $\underline{e}$ cancels out e and such sequences are reduced to 0.

1. $0/e = 0$
2. $\perp/e = \perp$
3. $(E_1 \wedge E_2)/e = ((E_1/e) \wedge (E_2/e))$
4. $(E_1 \vee E_2)/e = ((E_1/e) \vee (E_2/e))$
5. $(e \circ E)/e = E,$ if $e \notin \Gamma_E$
6. $D/e = D,$ if $e \notin \Gamma_D$
7. $(e' \circ E)/e = 0$, if $e \in \Gamma_E$
8. $(\underline{e} \circ E)/e = 0$

Figure 2.13 Calculating the guards in Singh's logic-based approach

Finally, events in the logic-based framework of [Sin97] produce notifications, which are incrementally collected or assimilated by the recipients. This leads to the simplification of their guards. The authors in [Sin97] define rules for assimilating messages; these are based on a '÷' operator which essentially embodies a set of 'proof rules' to reduce guards when an event occurs.

No implementation of the logic-based approach is described in [Sin97] although the rules for calculating the guards given in Fig. 2.13 should make this straightforward. Our interest in this approach stems from the fact it uses a logic to describe service coordination and capture temporal aspects of service execution (or not). It also considers coordination of services in a distributed manner, an aspect particularly relevant to multi-service transactions within a digital ecosystem.

2.3 A Formal Language for Long-running Transactions in DEs

In this section, we lay the foundations for a formal model of long-running transactions showing how the subtransactions (Local Coordinators and/or basic services) are orchestrated to achieve the goal of the transaction in question. In developing the formal model we have avoided following a specific formalism (such as CSP or CCS) so as not to be constrained from the outset by the limitations of a specific approach. In the first instance, this allows an understanding of long-running transactions that goes beyond that of *Sagas* [G-MS87] and we are not forced to consider a long-running transaction as a sequence of actions.

In our previous report, deliverable D3.1 of OPAALS [RMK07], we have described a distributed transaction model for the digital ecosystem paradigm, in which networked organisations are expected to engage in complex transactions involving a number of subtransactions (internal activities) which need to be coordinated locally in terms of the underlying service compositions. In other words, have opted for a open nested transaction model in which a transaction is understood to comprise a nested group of subtransactions whose actions (or service executions) do not necessarily have to be sequential, as in compensating CSP [BHF05] or the approach of [BMM05] discussed previously.

In this section we report on work in progress in providing a formal foundation for the proposed model of long-running transactions and the distributed orchestration of the underlying service compositions. Our formal semantics of long-running transactions is aimed at describing the behavioural patterns services should follow in order to guarantee successful commitment or compensation within the transaction flow manager. The proposed formal model for transactions uses ideas taken from a variety of theories for describing the behaviour of communicating systems, from Shields' vector languages [Shi79, Shi85, Shi97] to Mazurkiewicz traces [Maz77, Maz88] to event structures [NPW81] to process algebras [Mil80, Hoa85]. It draws upon a vector language-based description of behaviour, which allows monitoring or recording a number of communicating entities at the same time (groups of subtransactions), and has most recently been applied to modelling interactions between components of a (distributed) system in [Mos05]. This theory is adopted here to underpin the local coordination required for long-running multi-service transactions in a digital ecosystem.

In our model for long-running transactions in digital ecosystems, described in detail in D3.1 [RMK07], a transaction is represented by a tree structure that allows us to exemplify the local coordination that is required for the services involved to be performed in unison in accomplishing the goal prescribed by the transaction. In fact, at the heart of our transaction model are the Local Coordinators. This is where most complexities of a transactional environment are handled such as coordination, service orchestration, keeping logs for managing dependencies and implementing a recovery mechanism, but also this is where we would expect that handling the low bandwidth or dealing with the low processing power of some nodes in the underlying P2P network, would take place.

Based on the latest work on an extended service-oriented architecture for a business environment [YPH02], [Pap03], [PTD+06], five different types of coordinators have been considered in our model for, namely a data-oriented coordinator, a sequential process-oriented coordinator, a parallel process-oriented coordinator, a sequential-alternative coordinator a parallel-alternative coordinator and a delegation coordinator. Fig. 2.14 shows a transaction tree with four basic services whose order of execution is determined by the five coordinator types employed. We have adopted the notation of [PDH+96] extended with a symbol for the data-oriented coordinator (labelled by *d1* in the figure).

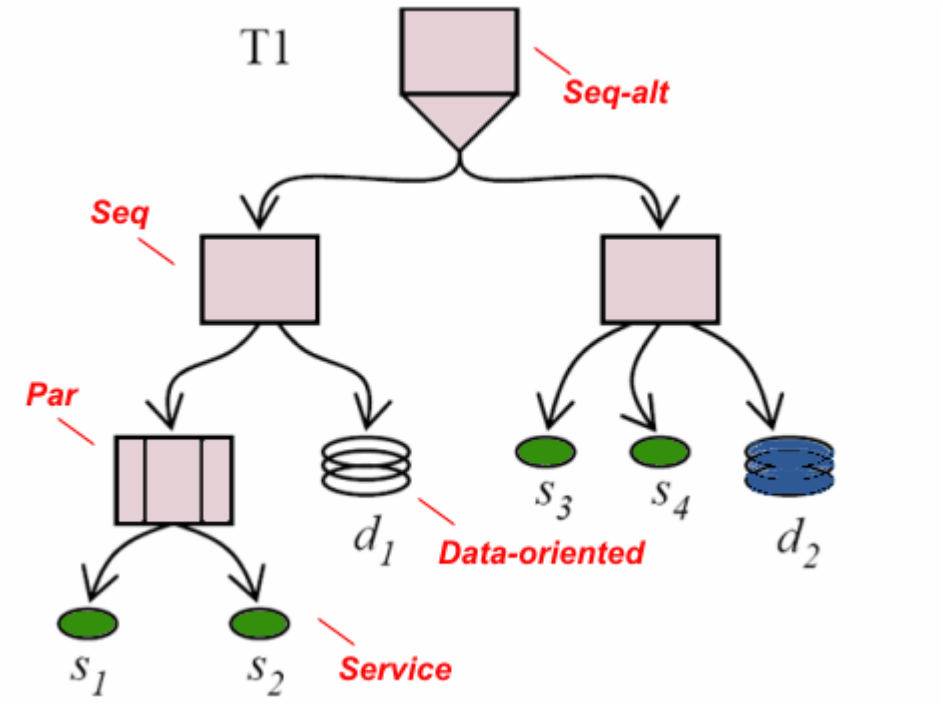


Figure 2.14 Transaction in a tree structure

The scenario described in Fig. 2.14 has also been used in deliverable D3.1 [RMK07] and is rather simple, nevertheless the transaction in question is complicated enough to illustrate the key ideas of our formal approach (and allows the reader to follow on from the description of the transaction model in D3.1). In the transaction tree of Fig. 2.14, s3 and s4 for example are children of a sequential coordinator and hence the service s4 can only be executed after s3. In other words the execution of s4 is dependent on the (successful) execution of s3. This sequential dependency may be due to the order of execution (e.g. book hotel after booking the flight) or due to a data dependency between s3 and s4, i.e. s4 has to use the results released by s3. This means, as a consequence, that if s3 is aborted, then s4 must also be aborted. In deliverable D3.1 we introduced the *Internal Dependency Graph* (IDG) for representing such dependencies. The dependency between s3 and s4 is shown in the IDG of Fig. 2.15(i).

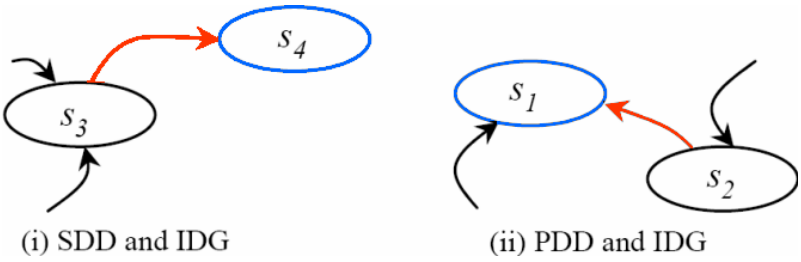


Figure 2.15 Internal Dependency Graph

In a highly dynamic and purely distributed environment such as a Digital Business Ecosystem, it is often the case that a sub-transaction requires access to a data item released (possibly as a partial result) by a sub-transaction belonging to a different transaction. In other words, dependencies may exist not only within a transaction but also between transactions (which may take place on different platforms). For example, consider the case of (compensatory) subtransactions that release partial results in a conditional commit state [PDH+96].

To capture such dependencies we introduced the *External Dependency Graph (*EDG) in D3.1 [RMK07]. This keeps track of dependencies between (services or coordinators of) different transactions. The log structure it provides can be used in recovery routines for running a compensating procedure. Figure 2.16 shows part of the EDG for the transaction trees T1 (of Figure 2.14) and T2. In this case, the data-oriented coordinators d1 and d2 of T1 release partial results that are required by d3 of T2.

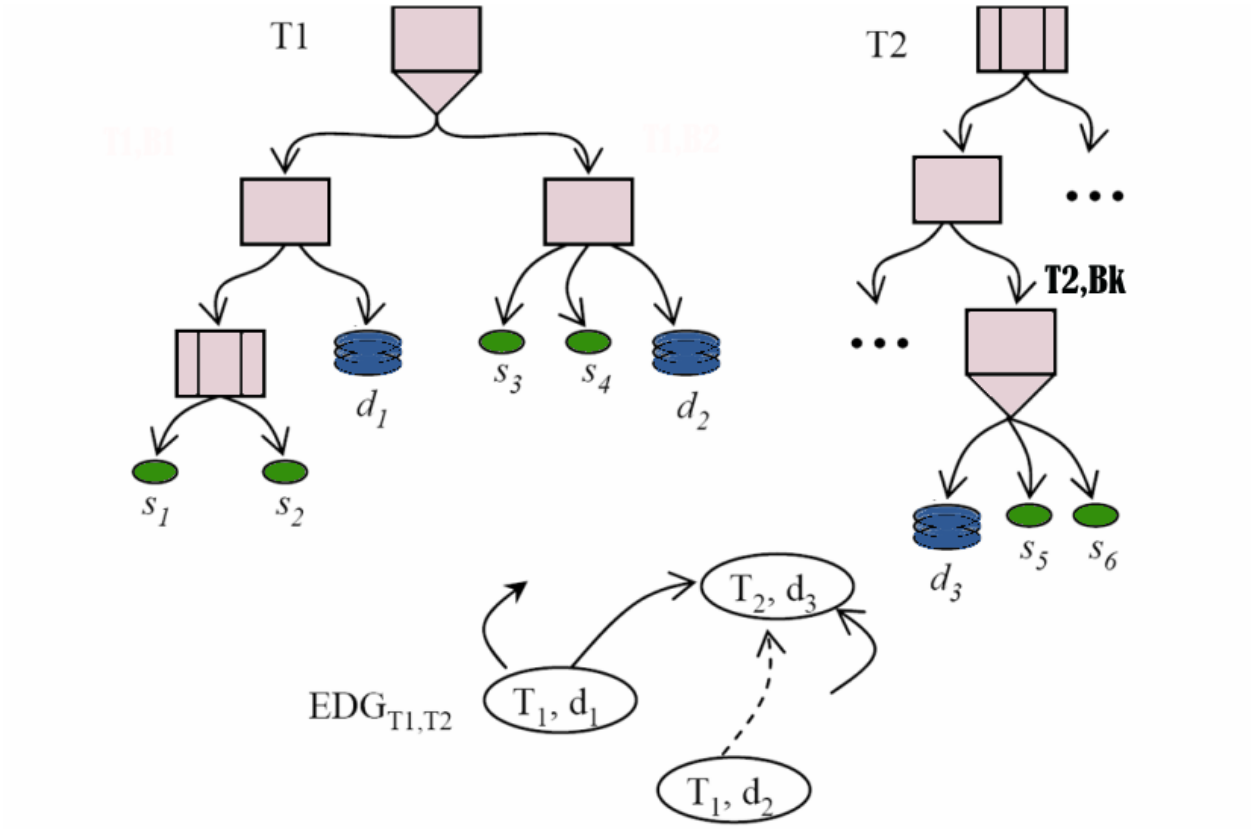


Fig. 2.16 EDG for releasing partial results between T1 and T2

Now, if for some reason d1 (or any other subtransactions on which d1 depends, for that matter) was aborted, then d3 should also be aborted along with any sub-transactions of T2 which depend on it. Based on the log information provided by the EDG and the corresponding transaction trees, we would like to recalculate d3 based on the data items released by d2 and defer from aborting (at least part of) transaction T2.

2.3.1 Transaction vectors: a language-based representation

We may now start to describe long-running transactions more formally. Apart from the dependencies there is a high degree of concurrency in a transactional environment. We will find the general theory of non-interleaving representation of parallel behaviour found in [Shi97] of great use in what follows. As mentioned before, our objective is to get a thorough understanding of the behaviour the underlying service compositions needs to exhibit for successful commit or compensation of the transaction as a whole.

In our behavioural model of a transaction it suffices to use formal notation for the leaves only. The aggregation coordinators (nodes) are manifested in the structure of the resulting formal construction, and there is no need for additional notation. A transaction T, then, is associated with a set of leaves L which consists of a set of basic services S, a set of data-oriented coordinators D and a set of delegation coordinators Dlg. Hence, $L = S \cup D \cup Dlg$. We further require that the sets S, D and Dlg are pairwise disjoint.

In this section we introduce a formal language for describing long-running transactions. 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 are offered in a digital ecosystem for business from different service providers and it is important that we defer from interfering

with the local state of the service execution. The service-oriented architecture for distributed transactions reinforces our interest in all environmentally observable actions inside and outside a transaction. That 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* [Sin97] or *actions* that may occur (on its subtransactions) upon activation, 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.

These actions take place on the leaves and therefore it seems appropriate to say that 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 $\mu(l)$, $l \in \mathrm{L}$, and require that $\bigcup_{l \in L} \mu(\mathrm{l}) \subseteq \mathrm{M}$.

In any behaviour of a transaction T, each subtransaction on the leaves will be activated and experience a sequence of actions formed over the corresponding set $\mu(l)$, $l \in \mathrm{L}$. This means that there a number of activation points within a transaction – essentially these are all its leaves. Following the idea that originates in Shields' vectors [Shi79], which was subsequently extended to a more general theory of communicating systems in [Shi97], we may describe the behaviour of a transaction by assigning such sequences to each of its leaves. This results in the so-called *transaction vectors* defined below.

Definition 1. (**Transaction vectors.**) Let T be a transaction. We define $\mathrm{V_T}$ to be the set of all functions $\underline{v}$: $\mathrm{L} \rightarrow \mathrm{M}^*$ such that $\underline{v}(l) \in \mu(l)^*$. We refer to elements of $\mathrm{V_T}$ as *transaction vectors*.

By $\mu(l)^*$ we denote the set of finite sequences over $\mu(l)$. Mathematically, the set $\mathrm{V_T}$ 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.

When an action occurs on a leaf of the transaction tree, that is to say when an action associated with some subtransaction takes place, it appears on a new transaction vector 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. 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 that of the second coordinate. Nothing has happened on the service corresponding to the third coordinate.

In this sense, each transaction vector provides a *snapshot* of behaviour in which the transaction has executed the actions appearing on the vector's coordinates – the vector tells us what actions have already occurred and on which part of the transaction tree.

This vector-based description of behaviour allows recording the actions of a transaction as these occur on the multiple services involved in the execution of the transaction. 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.

It can be seen from the examples given above that there is already an ordering among actions on a particular subtransaction (e.g. s_1 followed by another s_1). This vector-based behavioural description of transactions can also capture the orderings between different subtransactions, which amounts to actions appearing on different vector coordinates. This requires however a more careful consideration of the mathematical properties of such vectors which we briefly describe in the following section.

Before examining the mathematical properties of our construction so far, we introduce a specific kind of transaction vectors, which is used in our model to describe actions (events or activations) within a transaction.

Definition 2. (Column vectors.) Let T be a transaction and V_T its set of transaction vectors. We define

$$A_T = \{\underline{a} \in V_T \setminus \{\underline{\Lambda}_T\} : l \in L \Rightarrow |\, \underline{a}(l) \,| \leq 1\}$$

where $|\, x \,|$ denotes the length of sequence x. We refer to elements of A_T as *column vectors*.

Thus, the vectors of Definition 2 are themselves transaction vectors, but have the additional constraint that each of their coordinates is either the empty sequence or a single action. For example, the vector (s_1, Λ, Λ) represents the occurrence of an action s_1 on the sub-transaction associated with the first coordinate.

We will use the term *transaction language* to refer to a subset V of all possible vectors V_T formed over a given transaction T. Hence, a transaction T comes with a language V, where $V \subseteq V_T$. The idea is that the particular set of transaction vectors for a specific transaction expresses the ordering constraints necessary in the corresponding service orchestration.

2.3.2 Order-theoretic properties of transaction vectors

In what follows we describe the basic order-theoretic properties of transaction vectors since this is what allows us to define operations on vectors. These are important in determining the coordination of the transaction in terms of its underlying service invocations. We will see how the order structure of sets of such vectors expresses ordering constraints on actions inferred by the execution of the various subtransactions inside a long-running transaction.

In what follows we describe the basic mathematical properties of vectors. A detailed mathematical treatment of vectors can be found in [Shi97]. We have seen that transaction vectors are essentially tuples of sequences. This can be exploited in defining operations on the vectors in terms of well-known operations on sequences.

First, let us establish our notation. If x and z are sequences, we write x.z for the concatenation of x and z. As is well known this operation on sequences is associative with identity Λ, where Λ denotes the empty sequence. We also have a partial order on sequences given by $x \leq z$ if and only if there exists a sequence y such that x.y = z, and this partial order has a bottom element Λ. It is also well-known that the operation '.' is cancellative, which means that if $x \leq z$, then the sequence y such that x.y = z is unique. We shall denote

this sequence by z / x. Finally, recall that if x, y, z are sequences such that x, y $\leq$ z, then either x $\leq$ y or y $\leq$ x.

We may now lift these well-known operations on sequences onto transaction vectors. This is done formally in the following definition.

Definition 3. (**Operations on vectors.**) For $\underline{u}, \underline{v} \in V_T$, we define

- $\underline{u}.\underline{v}$ to be the unique vector $\underline{w}$ such that $\underline{w}(l) = \underline{u}(l).\underline{v}(l)$, for each $l \in L$ (*concatenation*)

- $\underline{u} \leq \underline{v}$ iff $\underline{u}(l) \leq \underline{v}(l)$, for each $l \in L$ (*prefix ordering*)

- $glb(\underline{u},\underline{v})$ to be the vector $\underline{w}$ such that $\underline{w}(l)=\min(\underline{u}(l),\underline{v}(l))$, for each $l \in L$

- $lub(\underline{u},\underline{v})$ (if it exists) to be the vector $\underline{w}$ such that $\underline{w}(l)=\max(\underline{u}(l),\underline{v}(l))$, for each $l \in L$

- if $\underline{u} \leq \underline{v}$, then we define $\underline{v} / \underline{u}$ to be the unique $\underline{z} \in V_T$ such that $\underline{u}.\underline{z} = \underline{v}$ (*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,

$$(s_1 s_3, s_2, \Lambda).(\Lambda, s_4, \Lambda) = (s_1 s_3, s_2 s_4, \Lambda)$$

Transaction vectors can be seen to be built up from the empty vector $\underline{\Lambda}_T$ by a series of concatenations with column vectors (Definition 2) 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*.

For example, consider a transaction with three leaves (basic services) in which a service s1 is intended to execute first, then a service s2 (which uses results of s1, and is thus dependent on s1) and after that execution continues with another service s3. This kind of (sequential) behaviour can be modelled by a series of concatenations. We assume the actions are labelled by the service name here, so we have actions a1 = (s1, Λ, Λ) for service invocation s1, a2 = (Λ, s2, Λ) for service invocation s2, and a3 = (Λ, Λ, s3) for service invocation s3.

Initially nothing has happened. This is described by the empty vector $\underline{\Lambda}_T$ = (Λ, Λ, Λ).

Then, s1 occurs. This is described in a vector $\underline{v}$ which is obtained by concatenating $\underline{\Lambda}_T$ with the column vector representing the action s1. Hence, we have

$$\underline{\Lambda}_T.\underline{a}_1 = (\Lambda, \Lambda, \Lambda).(s1, \Lambda, \Lambda) = (s1, \Lambda, \Lambda) = \underline{v}$$

Then, s2 occurs. This is described in a vector $\underline{u}$ which is obtained by concatenating $\underline{v}$ (the latest behaviour we have) with the column vector representing the corresponding action s2, here $\underline{a}_2$. Hence, we have

$$\underline{v}.\underline{a}_2 = (s1, \Lambda, \Lambda). (\Lambda, s2, \Lambda) = (s1, s2, \Lambda) = \underline{u}$$

Then, s3 occurs. This is described in yet another vector $\underline{w}$ which is obtained by concatenating $\underline{u}$ (the latest behaviour we have) with the column vector representing the corresponding action s3, here $\underline{a}_3$. Hence, we have

$$\underline{u}.\underline{a}_3 = (s1, s2, \Lambda).\ (\Lambda, \Lambda, s3) = (s1, s2, s3) = \underline{w}$$

In Section 2.3.3 we shall impose conditions on transaction languages that ensure they comprise transaction vectors which are obtained in this way, and the coordination of the underlying services it determines corresponds to intended behaviour of the transaction only.

The ordering amongst vectors is defined in terms of the usual prefix ordering operation on sequences appearing on their coordinates. For example,

$$(s_1, s_2, \Lambda) \leq (s_1 s_3, s_2, \Lambda) \text{ since } s_1 \leq s_1 s_3 \text{ and } s_2 \leq s_2 \text{ and } \Lambda \leq \Lambda$$

In other words, the vector $\underline{v}$ '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. For example,

$$(s_1 s_3, s_2, \Lambda) \text{ and } (s_1 s_5, s_2, \Lambda)$$
or
$$(s_1, \Lambda, \Lambda) \text{ and } (\Lambda, s_2, \Lambda)$$

It turns out that such vectors describe either parallel or alternative behaviours of the transaction in question, and this will be further discussed in Section 2.3.4.

It is important to note that these two fundamental operations, *concatenation and prefix-ordering*, on transaction vectors are performed *coordinate-wise* in our model and this simplifies the mathematics of it and allows for relatively straightforward proofs.

The operations *glb*() and *lub*() of Definition 3 give the greatest lower bound and the least upper bound, respectively of $\underline{u}, \underline{v} \in V_T$, in the usual sense of lattices and domain theory [DaP90]. For example, for vectors $\underline{v}$ = (s1, s2, Λ) and $\underline{u}$ = (s1, Λ, s3) the *glb*($\underline{u}$, $\underline{v}$) is computed as follows,

$$glb(\underline{u}, \underline{v}) = glb(\ (s1, s2, \Lambda), (s1, \Lambda, s3)\ ) = (s1, \Lambda, \Lambda)$$

since

$$\min(\underline{u}(1), \underline{v}(1)) = s1$$
$$\min(\underline{u}(2), \underline{v}(2)) = \Lambda$$
$$\min(\underline{u}(3), \underline{v}(3)) = \Lambda$$

Similarly, their least upper bound *lub*($\underline{u}$, $\underline{v}$) is computed as follows,

$$lub(\underline{u}, \underline{v}) = lub(\ (s1, s2, \Lambda), (s1, \Lambda, s3)\ ) = (s1, s2, s3)$$

since

$$\max(\underline{u}(1), \underline{v}(1)) = s1$$
$$\max(\underline{u}(2), \underline{v}(2)) = s2$$
$$\max(\underline{u}(3), \underline{v}(3)) = s3$$

These operations are central to the treatment of concurrency in our approach and also have an important role to play in defining the properties that ensure the well-formedness of the behavioural description, as will be discussed in Section 2.3.3.

The right cancellation operator '/' says that if $\underline{u}$ is a transaction vector describing an initial part of the behaviour described by $\underline{v}$ so that $\underline{u} \leq \underline{v}$, then $\underline{v} / \underline{u}$ is the 'continuation' of $\underline{u}$ that extends it to $\underline{v}$. This operation is central to the treatment of compensations in our approach. It is also particularly useful, together with the ordering '$\lhd$' (cf Definition 6), in deriving a transition relation that allows to associate the vector-based description of behaviour with automata and asynchronous transition systems [Shi85], in giving a state-based description of the interactions involved [MSK05].

It can be shown (by an adaptation of the proof found in [Mos05], which is in turn based on that originally perceived in [Shi97]) that a set of transaction vectors equipped with the operations of concatenation and prefix ordering of Definition 3 forms a monoid[1] and a partial order. $\underline{\Lambda}_T$ is used to denote the empty vector which has the empty sequence on each of its coordinates.

Proposition 1. A set of transaction vectors V_T is

1. a monoid under '.' and identity $\underline{\Lambda}_T$
2. a partial order under $\leq$ and bottom element $\underline{\Lambda}_T$

Proof.

For (1), it suffices to show that V_T is closed under '.' and that '.' is associative. We argue coordinate-wise. Let $\underline{u}, \underline{v} \in V_T$ and $l \in L_T$. Since $\underline{u}(l), \underline{v}(l) \in \mu(l)^*$ we have that $(\underline{u}.\underline{v})(l) \in \mu(l)^*$. Hence, $\underline{u}, \underline{v} \in V_T$, proving that V_T is closed under '.'. Now, for associativity, if $\underline{u}, \underline{v}, \underline{w} \in V_T$, then for each $l \in L_T$ we have

$$(\underline{u}.(\underline{v}.\underline{w}))(l) = \underline{u}(l).(\underline{v}.\underline{w})(l) = (\underline{u}(l).\underline{v}(l)).\underline{w}(l) = (\underline{u}.\underline{v})(l).\underline{w}(l) = ((\underline{u}.\underline{v}).\underline{w})(l)$$

Since $(\underline{u}.(\underline{v}.\underline{w}))(l) = ((\underline{u}.\underline{v}).\underline{w})(l)$, for all $l \in L_T$, we have that $\underline{u}.(\underline{v}.\underline{w}) = (\underline{u}.\underline{v}).\underline{w}$. so '.' is associative.

For (2), we need to show that '$\leq$' is reflexive, antisymmetric and transitive. Again, we argue coordinate-wise. Let $\underline{u}, \underline{v}, \underline{w} \in V_T$. Since $\underline{u}(l) \leq \underline{u}(l)$, for all $l \in L_T$, we have that $\underline{u} \leq \underline{u}$, giving reflexivity. If $\underline{u} \leq \underline{v}$ and also $\underline{v} \leq \underline{u}$, then $\underline{u}(l) \leq \underline{v}(l)$, for all $l \in L_T$, and $\underline{v}(l) \leq \underline{u}(l)$, for all $l \in L_T$, so we deduce that $\underline{u}(l) = \underline{v}(l)$, for all $l \in L_T$, which implies that $\underline{u} = \underline{v}$, proving antisymmetry. Finally, if $\underline{u} \leq \underline{v}$ and $\underline{v} \leq \underline{w}$, then $\underline{u}(l) \leq \underline{v}(l)$, for all $l \in L_T$, and $\underline{v}(l) \leq \underline{w}(l)$, for all $l \in L_T$, so $\underline{u}(l) \leq \underline{w}(l)$, for all $l \in L_T$, which in turn implies that $\underline{u} \leq \underline{v}$, proving transitivity. □

We note that a transaction language $V \subseteq V_T$ is not a monoid in general.

The incomparable vectors in the partial order $(V_T, \leq)$ allow to introduce a notion of independence between transaction vectors, which is central to expressing true-concurrency within our model. This builds on earlier work on describing parallel behaviour in Shields' *behaviour vectors* [Shi97] where the notion of independence found in *Mazurkiewicz traces* [Maz88] is lifted onto vectors. This development is the topic of Section 2.3.5 where we are concerned with modelling concurrent actions of a long-running transaction.

2.3.3 Well-formedness of the behavioural description of a transaction

In describing the behaviour of transaction we are interested in the actions (activations) on its sub-transactions. These are captured in our model using column vectors (Definition 2). Thus, instead of considering all possible transaction vectors we would like to be concerned with those obtained by concatenations with column vectors only. This gives us the behaviour of the transaction in terms of

[1] Recall that a monoid is a semi-group with identity.

activations or actions of its sub-transactions and can be used to enforce the coordination of the underlying services.

We have seen that transaction vectors are obtained by coordinate-wise concatenation (Definition 3), for example

$$(x_1, x_2, x_3).(y_1, y_2, y_3) = (x_1 y_1, x_2 y_2, x_3 y_3)$$

In such a behavioural description of a transaction, transaction vectors can be seen to be built up from the empty vector by a series of concatenations with the column vectors [Mos05], each of whose coordinates is either empty or contains a single event/action.

For example, the column vector $\underline{a} = (s_1, \Lambda, \Lambda)$ represents the activation of the leaf corresponding to the first coordinate. If s1 is intended to occur only after both s3 and s4 have, then this is described in the transaction vector $\underline{v} = (s_1, s_3, s_4)$ which is obtained as

$$\underline{u}.\underline{a} = (\Lambda, s_3, s_4).(s_1, \Lambda, \Lambda) = (s_1, s_3, s_4)$$

The study of vector languages in [Shi97, ShM04] shows that in order to ensure that vectors associated with a transaction are the result of concatenations with column vectors only, the set of transaction vectors must satisfy certain properties, namely *discreteness* and *local left-closure*. We introduce these properties next.

In describing the behaviour of a transaction in terms of the coordination of its subtransactions, we want to capture the fact that a system's computations always have a starting point, and ensure that only a finite number of events may occur within finite time. This turns out to be the case if whenever two vectors describe an earlier part of behaviour than a third, also in the set, then their least upper and greatest lower bounds are also in the set. This is formally put in the following definition.

Definition 4. (**Discreteness.**) Let $V \subseteq V_T$, then V is *discrete* if and only if, $\underline{\Lambda}_T \in V$ and whenever $\underline{u}, \underline{v}, \underline{w} \in V$ such that $\underline{u}, \underline{v} \leq \underline{w}$ then

(i) $lub(\underline{u}, \underline{v}) \in V$

(ii) $glb(\underline{u}, \underline{v}) \in V$

Note that $lub(\underline{u}, \underline{v}) \in V$ is understood as asserting that $lub(\underline{u}, \underline{v})$ is defined, i.e. the least upper bound of $\underline{u}, \underline{v}$ exists. This property builds on the notion of consistently complete subsets, as discussed in [Shi97], and further requires that the least upper and greatest lower bounds belong to the set.

It can be seen that discreteness imposes a finiteness constraint in the sense that it excludes infinite ascending or descending chains of actions with respect to time ordering. It ensures that situations like those resulting in Zeno-type paradoxes will never arise. The famous Zeno paradoxes, in which the philosopher seeks to demonstrate the impossibility of motion, are examples of a non-discrete representation of system behaviour[2].

Consider a transaction T which involves the execution of two basic services, and let

$$V_T = \{(\Lambda, \Lambda), (s_1 s_1, \Lambda), (\Lambda, s_2 s_2), (s_1 s_1, s_2 s_2)\}$$

[2] Zeno's paradox with arrow and that involving Achilles and the tortoise are discussed in view of computer science in [Shi97]. The conclusion drawn from Zeno's arguments in this case is not that motion is impossible, but that the behavioural description used is not *discrete*.

We observe that V_T is discrete (by checking against Definition 4). Indeed, V_T is a lattice in which greatest lower bounds are computed coordinate-wise. However, the corresponding transaction language has the counter-intuitive property that although four actions have occurred, there are only two elements, namely (s_1s_1, Λ) and (Λ, s_2s_2) to represent that portion of behaviour. The two vectors represent the second of the two actions only, at each service. We would like to eliminate such situations.

In order to obtain a precise description of discrete behaviour, we further require that every occurrence of an action (e.g. service invocation, partial result, commitment) is recorded in the set of vectors associated with the transaction. This guarantees that any earlier part of behaviour is itself a behaviour and motivates the following definition.

Definition 5. (**Local left-closure.**) Let $V \subseteq V_T$, $l \in L$ and $x \in \beta(l)^*$. Then, V is locally left-closed if and only if, whenever $\underline{v} \in V$ and $\Lambda < x \leq \underline{v}(l)$, then there exists $\underline{u} \in V$ such that $\underline{u} \leq \underline{v}$ and $\underline{u}(l) = x$.

The above definition says that whenever there is a sequence of actions on some sub-transaction (or Local Coordinator) which is less or equal than some other sequence appearing in some transaction vector in V, then there is some other vector in V which describes an earlier part of behaviour and has that sequence on the corresponding coordinate. In fact, 'local' comes from the fact the property is considered at the vector coordinate level and thus applies to individual subtransactions or Local Coordinators and 'left-closure' reflects the fact that earlier parts of a given behaviour are themselves behaviours.

To establish some terminology for the sequel, we say that the set of vectors $V \subseteq V_T$ associated with a transaction T is *normal* if and only if it is locally left-closed and discrete. This reflects the fact that the guarantees that accrue from these properties are embedded in the behaviour of the corresponding transaction.

Effectively, the local left-closure property is intended to resolve ambiguities that may arise from not having enough vectors in the transaction language to describe the course of the behaviour in question; not the start or the end, but the 'gaps' in between, as demonstrated in the example given prior to Definition 5. This requires that every occurrence of an event is 'recorded' in the language of the transaction. This implies the presence of a distinct *prime* element in V for each occurrence of an action, and on each appropriate leaf of the transaction tree.

Primes play a central role in the more general theory of parallelism [Shi97] and in particular with respect to associating vector languages with behavioural presentations [Shi88] which are order-theoretic objects used to determine the temporal relation between occurrences of actions. For the purposes of the present report, and the adaptation of this theory in deriving a formal model for long-running transactions, it suffices to understand that, in this context, the notion of prime refers to transaction vectors which have a unique other vector immediately beneath them. Such an ordering among vectors in a transaction language V is based on the relation '$\lhd$' which we define next.

Definition 6 (**Cover.**) Suppose that $\underline{u}, \underline{v} \in V \subseteq V_T$. We shall say that $\underline{v}$ covers $\underline{u}$ in V, and we shall write $\underline{u} \lhd_V \underline{v}$, if

1. $\underline{u} \leq \underline{v}$ and $\underline{u} \neq \underline{v}$
2. If $\underline{z} \in$ V such that $\underline{u} \leq \underline{z} \leq \underline{v}$, then $\underline{z} = \underline{u} \ \vee \ \underline{z} = \underline{v}$

We will omit subscript V's when the language is clear from context.

Intuitively, the covers relation '$\lhd$' provides an ordering among elements of V, in which one is 'immediately beneath' the other, allowing no other vector in V to exist in between them.

The set $\mathrm{Pre}_V(\underline{v})$, defined for $\underline{v} \in \mathrm{V}$, is used to denote all vectors that are related to $\underline{v}$ by '$\lhd$' in V, or more simply the set of predecessors of the vector $\underline{v}$. Hence,

$$\mathrm{Pre}_V(\underline{v}) = \{\underline{u} \in \mathrm{V} \mid \underline{u} \lhd_V \underline{v}\}$$

A technical lemma shows that if $\underline{u}$ is an earlier part of $\underline{v}$, but not a predecessor of $\underline{v}$, then there is some predecessor of $\underline{v}$ that is larger than $\underline{u}$.

Lemma 1. Suppose that $\underline{u}, \underline{v} \in V \subseteq V_T$ such that $\underline{u} \leq \underline{v}$ and $\underline{u} \not\lhd \underline{v}$, then there exists $\underline{w} \in \mathrm{Pre}_V(\underline{v})$ such that $\underline{u} \leq \underline{w}$.

Proof.

Since the number of leaves in a transaction tree is finite, the set $\{\underline{z} \in \mathrm{V} \mid \underline{u} \leq \underline{z} \leq \underline{v}\}$ is finite. It is also non-empty since it contains $\underline{u}$. Thus, it has a maximal element. Let $\underline{w}$ be the maximal element of this set, then $\underline{w} \in \mathrm{Pre}_V(\underline{v})$. □

The following result relates the '$\lhd$' relation with the right-cancellation operator '/' of Definition 3 and is based on an analogous result found in [ShM04] (Proposition 5.2 in [ShM04]).

Proposition 2. Suppose that $\underline{u}, \underline{v} \in V \subseteq V_T$ and V is normal. If $\underline{u} \lhd \underline{v}$, then $\underline{v} / \underline{u} \in A_\mathrm{T}$.

Proof.

Since $\underline{u} \lhd \underline{v}$, we have that $\underline{v} / \underline{u} \neq \underline{\Lambda}_\mathrm{T}$. we want to show that $\underline{v} / \underline{u} \in \mathrm{A_T}$. If $\underline{v} / \underline{u} \notin \mathrm{A_T}$, then for some $l \in \mathrm{L}$, $\underline{v}(l) = \underline{u}(l).\mathrm{w}_1.\mathrm{w}_2$, where $\mathrm{w}_1, \mathrm{w}_2 \in > \Lambda$. By local left-closure, (and take $\underline{u}(l).\mathrm{w}_1.\mathrm{w}_2$ as x) there exists $\underline{w} \in \mathrm{V}$ such that $\underline{w} \leq \underline{v}$ and $\underline{w}(l) = \underline{u}(l).\mathrm{w}_1$.

Let $\underline{z} = lub(\underline{u}, \underline{w})$. Since V is discrete we have $\underline{z} \in \mathrm{V}$, and also $\underline{u} \leq \underline{z}$. But now $\underline{u}(l) < \underline{u}(l).\mathrm{w}_1 = \underline{w}(l) = \underline{z}(l)$ and hence, $\underline{u} \leq \underline{z}$. Also, $\underline{z}(l) = \underline{w}(l) = \underline{u}(l).\mathrm{w}_1$ and hence, $\underline{z} \leq \underline{v}$. This implies that $\underline{u} < \underline{z} < \underline{v}$, which is a contradiction to $\underline{u} \lhd \underline{v}$. □

This important result establishes that what takes a transaction vector and extends it to its successor is a column vector, representing an action of the transaction. This means that vectors in a normal transaction language are built by a series of concatenations with column vectors – in other words, the behavioural description is constructed by considering slices of behaviours that arise through the occurrence of actions as determined by the subtransactions of a transaction.

To anticipate further, these results will be useful in the development concerning the *t*-decompositions in our mathematical framework, which are central to the handling of compensations in our approach. This is the topic of Section 2.3.7 and 2.3.8.

In fact, discreteness and local left-closure ensure the well-formedness of the behavioural description of a transaction in our model. The idea is that in checking against these properties we may determine whether the transaction will exhibit the desired behaviour when executed or on the contrary, other non-desirable scenarios of execution are still possible. This draws upon previous work on vector languages and UML sequence diagrams in [Mos05].

2.3.4 Sequential actions

The prefix ordering (Definition 3) among 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 $\underline{u} \leq \underline{v}$ in a transaction language means that $\underline{u}$ is an earlier part of behaviour leading to $\underline{v}$.

If in addition the transaction language is normal, i.e. discrete and locally left-closed, then we can say more than that. In particular, we have seen (Proposition 2) that whenever $\underline{v}$ covers $\underline{u}$ in a normal transaction language, then what takes $\underline{u}$ and 'stretches it up' to $\underline{v}$ is a column vector representing the occurrence of an action (in fact, the model can express the occurrence of a simultaneity class of actions, based on [Shi97], but we have abstained from going this far for the moment). This allows us to model dependent actions. That is, occurrence of one action depends on the previous occurrence of the other. Recall the example in Section 2.3.1 where service s3 feeds service s4. It is in this sense that we talk about actions occurring in sequence (one after the other).

Suppose that during a long-running transaction a series of actions have already been executed and the resulting behaviour (up to that point) is described by a transaction vector $\underline{u}$ = (a1, Λ, c1, Λ). Then, occurrence of $\underline{a}_1$ = (Λ, b1, Λ, Λ) followed by occurrence of $\underline{a}_2$ = (a2, Λ, Λ, d1) can be modelled by

- first, concatenating vector $\underline{u}$ with $\underline{a}_1$ and
- then concatenating the resulting vector $\underline{v}$ with $\underline{a}_2$

In terms of our mathematical framework this amounts to operations $\underline{u}.\ \underline{a}_1 = \underline{v}$ and then $\underline{v}.\ \underline{a}_2 = \underline{w}$.

Considering the Hasse diagram for the order structure of the corresponding transaction language V, where lines between vectors denote an ordering relation in which the topmost vector is greater than the lower one, this would result in the portion of the diagram shown in Figure 2.16.

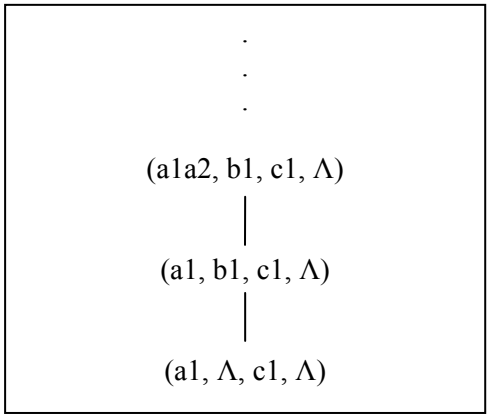


Figure 2.16 An action b1 followed by a2

It is important to make the observation that the actual ordering between actions appearing in different coordinates of a transaction vector is determined by context - by what other vectors are included in the language. In other words, the relationship between transaction vectors and associated order theoretic structures is very much dependent on what other vectors are in the set V (unlike the behaviour vectors in [Shi79], [Shi97] where this relationship is independent of context).

For instance, in transaction vector $\underline{v}$ = (a1, b1, c1, Λ) we may immediately derive that the action a1 has happened on the leaf corresponding to the first coordinate, action b1 has happened on the leaf corresponding to the second coordinate and action c1 has happened on the leaf corresponding to the third. To determine the relationship between these actions however, we need the rest of the language.

The following discussion illustrates this by means a small example.

Assume that V is given by the set

$$V = \{(\Lambda, \Lambda, \Lambda, \Lambda), (a1, \Lambda, c1, \Lambda), (a1, b1, c1, \Lambda), (a1a2, b1, c1, d1)\}$$

Notice that adding in $(\Lambda, \Lambda, \Lambda, \Lambda)$ is essential, and in this case is also sufficient for making V discrete and locally left-closed. Now the presence of $\underline{u} = (a1, \Lambda, c1, \Lambda)$ for which we have $\underline{u} \lhd \underline{v}$, tells us that action b1 on the subtransaction corresponding to the second coordinate occurs only after both actions a1 and c1 have taken place.

Now suppose that the intended behaviour of the transaction prescribed that a1 must occur before c1. This is captured in the corresponding transaction language by adding in the vector $(a1, \Lambda, \Lambda, \Lambda)$. In the resulting language

$$V = \{(\Lambda, \Lambda, \Lambda, \Lambda), (a1, \Lambda, \Lambda, \Lambda), (a1, \Lambda, c1, \Lambda), (a1, b1, c1, \Lambda), (a1a2, b1, c1, d1)\}$$

which continues to be normal. The presence of $\underline{w} = (a1, \Lambda, \Lambda, \Lambda)$ for which $\underline{w} \lhd \underline{u}$ dictates that a1 on the first coordinate occurs strictly before c1 does on the service corresponding to the third coordinate.

2.3.5 Concurrent actions

Our approach towards modelling concurrent actions, actions that can happen in parallel, draws upon the concepts in *Shields' vector languages* [Shi79, Shi97] and *Mazurkiewicz trace languages* [Maz77], [Maz88] where the ordering of concurrent events is considered subjective and thus is not distinguished, in contrast to CSP trace theory where it is assumed that observations are sequential in nature leading to the interpretation that concurrent events occur in either order.

For systems that exhibit concurrency, different external observers may disagree on the ordering of concurrent events. This may be seen more clearly in Einstein's famous thought-experiment[3] involving two trains travelling at constant speed in opposite directions along a pair of parallel tracks. Observers O1 and O2 are sitting in the middle of each train. A third observer O3 is sitting on the embankment. At a given moment, the two observers on the trains are on a line at right angles to the third observer. At that moment, two bolts of lightning strike on either end of the first train in such a way that O3 sees them strike at exactly the same time. Observer O1 travelling towards the light coming from the strike on the front end of the train he is on, sees that light before he sees the light of the strike on the rear end of the train. Observer O2 travelling towards the light coming from the strike on the rear end, sees that light before she sees the light coming from the strike on the front end.

Now from the point of view of observer O1 there are three distinct behaviours of the "system". One is when nothing has happened yet, another when he has seen the lightning bolt from the front end of the train, and another when he has seen both lightning bolts. Likewise, observer O2 has seen a behaviour in which nothing has happened yet, a behaviour where she has seen the lightning strike on the rear end and a behaviour where she has seen both lightning bolts. From the point of view of observer O3 there are only two distinct behaviours. One is when nothing has happened yet and the other is when both have. Thus, all four distinct behaviours can be observed for the same system; nothing has happened, one event has happened, the other event has happened, and both events have happened.

[3] This thought-experiment was given by A. Einstein to demonstrate the non-objectivity of contemporaneity in relativistic mechanics. It has been considered in view of concurrency in [Shi97] and our description of the experiment here is based on that.

The point to be made here is that observations on systems exhibiting concurrency largely depend on the relative position of the observer or the actual timing of execution. Such differences are non-objective and do not allow to infer the actual ordering between the events. On this basis, any particular ordering between concurrent events is irrelevant. On the contrary, the ordering between causally related events is objective (independent of the observer) and should be distinguished.

Returning to the treatment of concurrency within our formal model of transactions, this takes up on Mazurkiewicz traces [Maz77], [Maz88], 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.

Definition 7. (**Concurrent alphabet**) Let A denote a (finite) set. A *concurrent alphabet* is an ordered pair (A, ι) where the binary relation $\iota \subseteq A \times A$ satisfies

- $a \iota b \Rightarrow b \iota a$ (symmetry)
- $a \iota b \Rightarrow a \neq b$ (irreflexivity)

This definition gives an independence relation on action symbols from a set (alphabet) A. Symmetry requires that concurrency is always mutual while irreflexivity prohibits considering an action being concurrent with itself.

Transaction vectors are essentially tuples of sequences, as discussed before. Thus, we find it useful to consider the extension of the relation ι to sequences, based on [Maz77].

Given a concurrent alphabet (A, ι), a relation $\equiv_\iota^{(1)}$ can be defined on the set of all sequences over A, denoted by A*, by

$$x \equiv_\iota^{(1)} y \Rightarrow \exists\, u, v \in A^*, \exists\, a, b \in A \text{ such that } a \iota b \ \wedge\ x = uabv \ \wedge\ y = ubav$$

Let $\equiv_\iota$ be the reflexive, transitive closure of $\equiv_\iota^{(1)}$. By definition, $\equiv_\iota$ is an equivalence relation[4] on A*. The set of all sequences in A* that are related by $\equiv_\iota$ to a sequence $x \in A^*$ is called the equivalence class of x. We denote the equivalence class of a sequence $x \in A^*$ by $\langle x \rangle_\iota$. The set of equivalence classes of A with independence relation ι is denoted by $A_\iota^* = \{\langle x \rangle_\iota \mid x \in A^*\}$. Any subset L of A_ι^* is called a *Mazurkiewicz trace language*.

Therefore, the independence relation ι defined on the set A gives rise to an equivalence relation $\equiv_\iota$ on sequences formed over A. We make use of this construction in terms of sequences formed over the sets of actions μ(*l*), for each $l \in L$, associated with a transaction. It might be instructive at this point to revisit the definition of an independence relation given in compensating CSP [BHF05] discussed in Section 2.2.2.

Intuitively, the equivalence relation on sequences of actions equates all, and only those, sequences which differ in the order of adjacent independent actions. Such actions take place concurrently and the fact their ordering is irrelevant is reflected in the corresponding equivalent sequences. 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 μ(*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 μ(*l*), for each $l \in L$, no actions can be concurrent in the corresponding sequences μ(*l*)*, for each $l \in L$, which amounts to our understanding of sequential systems (e.g. as described by processes in CSP [Hoa85] and its extension with compensations in [BHF05]).

[4] Recall that an equivalence relation on a set is a binary relation that is symmetric, reflexive and transitive.

Drawing upon the extension of the independence relation ι to *behaviour vectors* in [Shi97], the notion of independence between actions in *Mazurkiewicz traces* can be readily interpreted into transaction vectors in our approach.

Definition 8. (**Independence**) For $\underline{u}, \underline{v} \in V \subseteq V_T$, we define

$$\underline{u} \text{ ind } \underline{v} \iff \forall\ l \in L :\ \underline{u}(l) > \Lambda \Rightarrow \underline{v}(l) = \Lambda$$

The 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 $\underline{u}$ and $\underline{v}$ may occur independently.

In the case of column vectors (recall Definition 2), independence captures the fact that actions appearing in one vector may occur independently of those appearing in the other. If in addition the vectors representing these actions are adjacent in an expression (of the series of concatenations that went into forming the corresponding transaction vectors), then the actions are concurrent. Thus, whenever two actions are independent and are both enabled (can both occur at some point, after some behaviour) then, their corresponding column vectors commute, i.e. $\underline{a}_1.\underline{a}_2 = \underline{a}_2.\underline{a}_1$, and in the resulting behaviour the two actions are concurrent. In fact, (A, ind) is a concurrent alphabet, in the sense of Definition 7.

For example, suppose that a transaction with 3 leaves has experienced a fragment of behaviour described by $\underline{u}$ = (s1, Λ, Λ) and after that may engage in $\underline{a}_1$ and $\underline{a}_2$ concurrently, where $\underline{a}_1$ = (Λ, s2, Λ) represents an invocation of service s2 and $\underline{a}_2$ = (Λ, Λ, s3) represents an invocation of service s3.

We make the observation that $\underline{a}_1$ ind $\underline{a}_2$ (by Definition 8) and consequently,

$$\underline{a}_1.\underline{a}_2 = (\Lambda, s2, \Lambda).(\Lambda, \Lambda, s3) = (\Lambda, s2, s3) = (\Lambda, \Lambda, s3).(\Lambda, s2, \Lambda) = \underline{a}_2.\underline{a}_1$$

Thus, we have $\underline{u}.\underline{a}_1.\underline{a}_2 = \underline{w} = \underline{u}.\underline{a}_2.\underline{a}_1$.

Indeed,

$$\underline{u}.\ \underline{a}_1 = (s1, \Lambda, \Lambda).(\Lambda, s2, \Lambda) = (s1, s2, \Lambda) = \underline{v}_1$$

and

$$\underline{v}_1.\ \underline{a}_2 = (s1, s2, \Lambda).(\Lambda, \Lambda, s3) = (s1, s2, s3) = \underline{w}$$

We also have that

$$\underline{u}.\ \underline{a}_2 = (s1, \Lambda, \Lambda).(\Lambda, \Lambda, s3) = (s1, \Lambda, s3) = \underline{v}_2$$

and

$$\underline{v}_2.\ \underline{a}_1 = (s1, \Lambda, s3).(\Lambda, s2, \Lambda) = (s1, s2, s3) = \underline{w}$$

In the resulting behaviour $\underline{w}$ the actions s1 and s3 are concurrent. The situation is depicted in the familiar diamond (or *lozenge*) appearing in Figure 2.17.

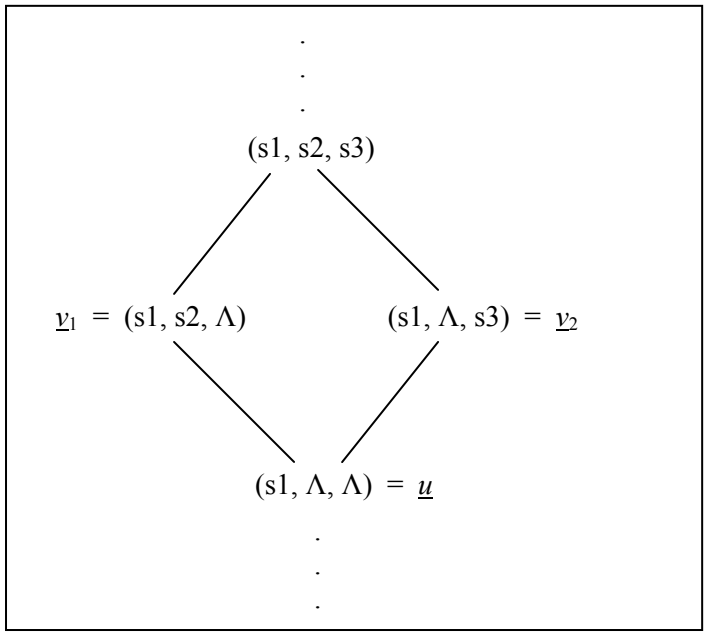


Figure 2.17 Concurrent actions in a transaction

Note that if the actions were not concurrent, then we would have the lower part of the diamond shape in the diagram but not the upper half. The upper half is obtained only when the column vectors corresponding to the actions in question commute (that is to say, equivalently, that they are independent) *and* represent actions that are both enabled after some given behaviour. Both these requirements have to be met for the actions to be concurrent. This is then reflected in the order structure of the corresponding transaction language by the presence of the vector forming the upper half of the diamond. This vector is the resulting common behaviour, after the concurrent actions have taken place. If either of these two requirements is violated, then the transaction would never exhibit the common behaviour described by $\underline{w}$. The point to be made here is that independence alone does not guarantee concurrency. (The case of non-independence is more obvious.)

As depicted in Figure 2.17, the transaction as a whole experiences both actions (on each appropriate leaf) and the ordering is irrelevant. The corresponding concatenations result in a unique transaction vector (sitting on the top of the diamond) in which both actions have occurred in *no particular order*. The incomparable transaction vectors in the middle of the diamond (i.e. $\underline{v}_1$, $\underline{v}_2$) represent behaviour arising during concurrent execution. These two vectors are bounded above by the vector in which both concurrent actions appear (i.e. $\underline{w}$).

In terms of the order theoretic properties of transaction vectors discussed in Section 2.3.2, this vector $\underline{w}$ is the least upper bound of the incomparable vectors. In the example of Figure 2.17, we have

$$lub(\underline{v}_1, \underline{v}_2) = lub(\ (s1, s2, \Lambda), (s1, \Lambda, s3)\ ) = (s1, s2, s3) = \underline{w}$$

Their greatest lower bound (sitting at the bottom of the diamond) is the vector in which none of the concurrent actions have occurred but are both available. In our example, we have

$$glb(\underline{v}_1, \underline{v}_2) = glb(\ (s1, s2, \Lambda), (s1, \Lambda, s3)\ ) = (s1, \Lambda, \Lambda) = \underline{u}$$

We note that this non-interleaving representation of concurrent behaviour manifests itself in the structure of the automata associated with this kind of vector languages, described in [ShM04], [MSK05], which build on the seminal work on asynchronous transition systems (ATS) [Shi85].

The fundamental difference in expressing concurrency should now be apparent. By departing from classic CSP concurrency, we are able to consider concurrency within a long-running transaction, and without the need to consider sequences of actions within a transaction as in compensating CSP [BHF05]. In CSP, and related process algebras, concurrency arises through composition. Here we have not yet been concerned with composing sequences from subtransactions of different transactions, though this may also produce concurrency. We are simply describing the case that subtransactions of the same transaction engage in concurrent actions, a phenomenon common in most B2B scenarios for example. The notion of composition within this vector language – based behavioural description has been described in [MoS04].

In what follows, we again discuss concurrent actions in connection to the context of the corresponding transaction language. Consider the transaction language

$$V = \{ (\Lambda, \Lambda, \Lambda), (s1, \Lambda, \Lambda), (s1, s2, \Lambda), (s1, \Lambda, s3), (s1, s2, s3) \}$$

It can be easily checked that V is discrete and locally left-closed. Its order structure is (in part) depicted in Figure 2.17. We have that $\underline{u} \lhd \underline{v}_1$ and $\underline{u} \lhd \underline{v}_1$. Also, we have $\underline{v}_1 \lhd \underline{w}$ and $\underline{v}_2 \lhd \underline{w}$. We have seen that the actions s2 on the second leaf and s3 on the third are concurrent.

Now consider the transaction language,

$$V = \{ (\Lambda, \Lambda, \Lambda), (s1, \Lambda, \Lambda), (s1, s2, s3) \}$$

In this language, which is also discrete and locally left-closed, the actions s2 on the second leaf and s3 on the third are simultaneous rather than concurrent. This is because the transaction vector $\underline{w}$ = (s1, s2, s3) in which both actions have taken place is obtained directly from $\underline{u}$ = (s1, Λ, Λ) in which neither of the actions have occurred yet. Hence, what takes $\underline{u}$ and stretches it up to $\underline{w}$ is the column vector $\underline{a}$ = (Λ, s2, s3) in which s2 and s3 are simultaneous actions. This case can be understood as cutting through the diamond of Figure 2.17.

Next, consider the transaction language

$$V = \{ (\Lambda, \Lambda, \Lambda), (s1, \Lambda, \Lambda), (s1, s2, \Lambda), (s1, \Lambda, s3) \}$$

In this language, which is also discrete and locally left-closed, the actions s2 and s3 are neither concurrent nor simultaneous. The transaction in this case, after doing s1 on the leaf corresponding to the first coordinate, has a choice between doing s2 on the second coordinate or s3 on the third. This case can be understood as having only the lower half of the diamond in Figure 2.17, and brings about the issue of alternative actions and mutual exclusion in a long-running transaction. This is discussed in the following section.

2.3.6 Alternative actions

Based on the prefix ordering between transaction vectors in the set V we may also model a choice between actions. That is, actions which are mutually exclusive in that occurrence of one excludes occurrence of the other.

In discussing concurrent actions in a long-running transaction, we saw that the two incomparable transaction vectors in the middle of the diamond represent concurrent behaviour. The fact that the two

incomparable vectors are in the middle of the diamond implies that they are bounded above in the set (by the transaction vector sitting on top of the diamond).

Whenever this latter requirement does not hold we may talk about events in conflict. In terms of pictures and associated Hasse diagrams, we are essentially getting rid of the upper part of the diamond and keeping the lower part, the branches of which represent a choice between doing one or the other action. In effect, this amounts to ensuring that the behaviours they represent is not (an early) part of the same behaviour. Therefore, in what follows we examine when two transaction vectors are not bounded above in a component language.

Let us first consider the case where the column vectors in question are not independent. Then, they do not agree on the non-empty coordinates corresponding to the same leaves of the transaction tree. This entails that there is no causality between the two and at the same time it is not possible for both of them to occur (since they engage the same leaf of the tree).

For example, the actions represented by $\underline{a}_1 = (s1, \Lambda, \Lambda)$ and $\underline{a}_2 = (s4, \Lambda, \Lambda)$ could both be available (or possible to occur) when the transaction has already exhibited the behaviour described by a transaction vector, say, $\underline{u}$. In other words, after $\underline{u}$ the transaction may engage in either $\underline{a}_1$ or $\underline{a}_2$. Note that it cannot do both since s1 and s4 ($s1 \neq s4$) are actions associated with the same service (the one corresponding to the first coordinate). Considering the Hasse diagram for the order structure of the corresponding transaction language, this situation would result in the fragment of the diagram shown in Figure 2.18.

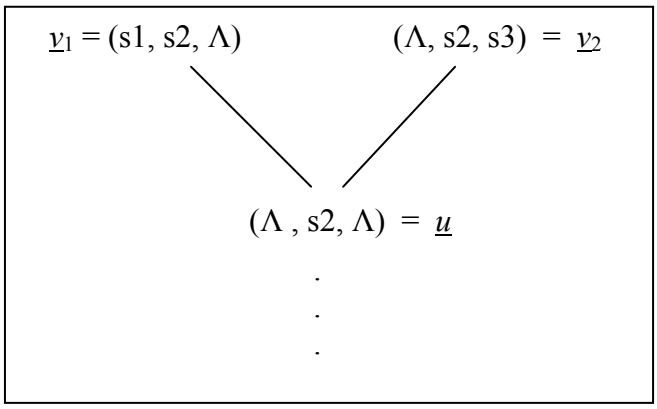


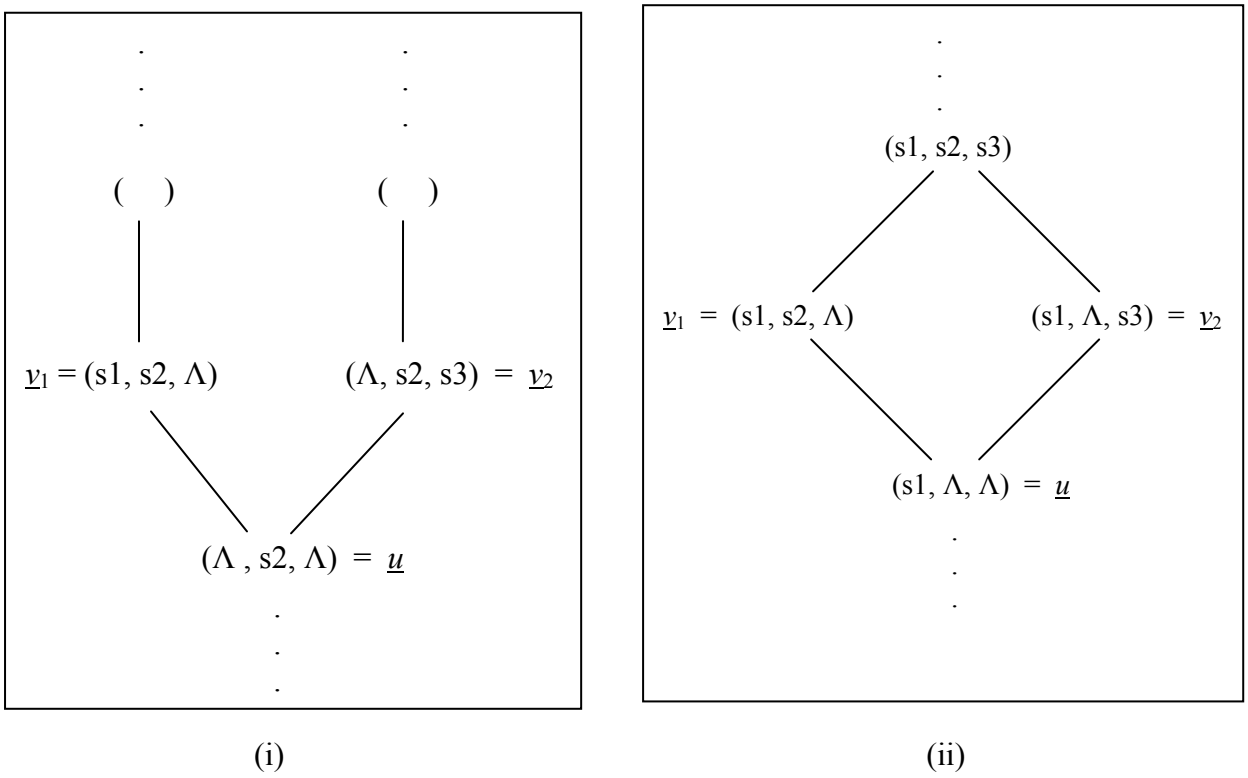
Figure 2.18 Alternative actions in a transaction

In further explanation, $\underline{v}_1 = (s1, s2, \Lambda)$ is the behaviour resulting from occurrence of $\underline{a}_1$ while the vector $\underline{v}_2 = (\Lambda, s2, s3)$ is the behaviour resulting from occurrence of $\underline{a}_2$, after $\underline{u}$. In terms of our mathematical framework, we have $\underline{u}.\underline{a}_1 = \underline{v}_1$ and $\underline{u}.\underline{a}_2 = \underline{v}_2$ but only one of these behaviours may take place during an execution of the long-running transaction in question.

We now turn our attention to actions whose corresponding column vectors are independent (recall Definition 8. This case is a bit more subtle. In principle, independent column vectors represent actions which are in no way related to each other. For example, consider the actions given by column vectors $\underline{a}_1 = (s1, \Lambda, \Lambda)$ and $\underline{a}_2 = (\Lambda, \Lambda, s3)$. If they are both offered after the transaction has engaged in behaviour described by vector $\underline{u}$, then they represent a choice between doing s1 on the leaf corresponding to the first coordinate and action s3 on the leaf corresponding to the third coordinate. Unless they are bounded above!

To ensure that the two independent events are not bounded above, effectively, that they are not part of a subsequent common behaviour, they must not occur consecutively. In other words, the actions succeeding $\underline{a}_1$ must not be $\underline{a}_2$ and, dually, the action succeeding $\underline{a}_2$ (on the other branch) must not be $\underline{a}_1$. Otherwise, they lead to a common behaviour $\underline{w}$ which inadvertently bounds $\underline{v}_1$ and $\underline{v}_2$ (forcing them to be concurrent as discussed before).

The situation is depicted in Figure 2.19(i) where the actions s1 and s3 are alternative. Compare with Figure 2.19(ii) where the actions s1 and s3 are concurrent.



(i) (ii)

Figure 2.19 Actions s1 and s3 are concurrent in (ii) but not in (i)

2.3.7 Compensation in transaction vectors

In this section we describe work in progress on the handling of compensations in our formal model for long-running transactions. The ideas presented in this section draw upon the analysis of the mathematical properties of vectors found in a number of texts, e.g [Shi97] and [ShM04] as a starting point. We have seen that the occurrence of an action is recorded in the vector language description by (coordinate-wise) concatenation of the existing vector (or vectors), describing the behaviour of the transaction before the action occurred, with the column vector representing the action in question.

We have also seen (Definition 3) that the right-cancellation operator '/' on vectors can be used to isolate the behaviour that arises in between vectors, so that if $\underline{u}$ is a transaction vector describing an initial part of

the behaviour described by $\underline{v}$ so that $\underline{u} \leq \underline{v}$, then $\underline{v} / \underline{u}$ is the 'continuation' of $\underline{u}$ that extends it to $\underline{v}$. Further, in a normal (discrete and locally left-closed) transaction language we have seen (recall Proposition 2)) that if $\underline{u}$ is an immediate predecessor of $\underline{v}$, i.e. $\underline{u} \lhd \underline{v}$ in the sense of Definition 6, then the application of the right-cancellation operator on $\underline{v}$ produces a vector which is a column vector, i.e. it isolates the last action(s) that went into obtaining $\underline{v}$ from $\underline{u}$.

The following result shows that the column vector $\underline{v} / \underline{u}$ is unique.

Lemma 2. Suppose that $\underline{v}, \underline{w} \in V \subseteq V_T$ such that $\underline{v} \leq \underline{w}$, then there exists unique vector $\underline{u} \in$ V such that $\underline{v}.\underline{u} = \underline{w}$. We denote this element by $\underline{w} / \underline{v}$.

Proof.

For each $l \in$ L, define $\underline{u}(l) = \underline{w}(l).\underline{v}(l)$. We have, for all $l \in$ L,

$$(\underline{v}.\underline{u})(l) = \underline{v}(l).(\underline{w}(l) / \underline{v}(l)) = \underline{w}(l)$$

so $\underline{v}.\underline{u} = \underline{w}$. Furthermore, if $\underline{u}' \in$ V such that $\underline{v}.\underline{u}' = \underline{w}$, then for each $l \in$ L, $\underline{v}(l).\underline{u}(l) = \underline{w}(l)$, so that

$$\underline{u}'(l) = \underline{w}(l) / \underline{v}(l) = \underline{u}(l), \text{ and } \underline{u}' = \underline{u},$$

establishing uniqueness. □

This result together with the main result of Proposition 2 in Section 2.3.3 allows us to define compensations by using the right-cancellation operator on vectors that differ only in an action (or, more generally, in a simultaneity class of actions). This can be done since the application of '/' on a vector undoes the last action that took place in obtaining the behaviour described by that vector.

In this way our approach uses (coordinate-wise) concatenation to model the occurrence of an action (the activation of a service invocation point in the transaction), by

$$\underline{u} \xrightarrow{a} \underline{v}$$

Instead of introducing separate notation and associated semantics for cancelling actions that have already taken place, the idea is to use right-cancellation on vectors to perform the compensation action, hence,

$$\underline{u} \xleftarrow{\underline{v}/\underline{u}} \underline{v}$$

which will be invoked if a failure later in the long-running transaction makes it necessary. In other words, the application of the right-cancellation operation on a vector $\underline{v}$ obtained by $\underline{u}$, produces $\underline{u}$, since

$$\underline{v} / (\underline{v} / \underline{u}) = \underline{u}$$

In what follows we show that compensations performed in this way will cancel all forward actions (modelled using concatenation on vectors), leaving only vectors where actions for which their compensations have also been executed are no longer visible in the end result.

Intrinsic to the development of the theory for compensations in our vector model is the fact that vectors describing the behaviour of a long-running transaction can be seen to be built up from the empty vector by a series of concatenations with column vectors representing appropriate actions. Therefore, vectors are essentially formed by a series of concatenations with column vectors. We find it useful to describe this more formally now, since the subsequent application of the right-cancellation would remove (or undo) each action in turn.

We have seen in Proposition 2, which is concerned with building up the vectors, the interaction between '$\lhd$' and '/' – that is, if $\underline{u} \lhd \underline{v}$ in a normal transaction language, then $\underline{v} / \underline{u}$ is an action, represented by a column vector (Definition 2). The following proposition is based on a result in [ShM04] for component languages and states a similar result for the interaction between '$\lhd$' and '.'.

Proposition 3. Suppose that $V \subseteq V_T$ is a normal transaction language, then

1. $\underline{\Lambda}_T \in V$
2. If $\underline{u}, \underline{v} \in V$ such that $\underline{u} \lhd \underline{v}$, then there exists $\underline{a} \in A_T$ such that $\underline{v} = \underline{u}.\underline{a}$.

Proof.

For (1). $V \neq \varnothing$, so there exists $\underline{v} \in V$. By local left-closure (Definition 6) of V, for each leaf $l \in L$, there exists $\underline{u}_l \in V$ such that $\underline{u}_l \leq \underline{v}$ and $\underline{u}_l(l) = \Lambda$. By discreteness (Definition 5) of V, the greatest lower bound of the $\underline{u}_l$, which must be $\underline{\Lambda}_T$, belongs to V.

For (2). Let $\underline{a} = \underline{v} / \underline{u}$, so that $\underline{v} = \underline{u}.\underline{a}$. This means that $\underline{a} \neq \underline{\Lambda}_T$. If $|\, \underline{a}(l) \,| > 1$, then we can find x, $y \in \mu(l)^*$ such that x, $y \neq \Lambda$ and $\underline{a} = x.y$. By local left-closure, there exists $\underline{x} \in V$ such that $\underline{x} \leq \underline{v}$ and $\underline{x}(l) = \underline{u}(l).x$. Let $\underline{w} = glb(\underline{u},\underline{x})$. We have that $\underline{w}$ exists, $\underline{w} \leq \underline{v}$ and $\underline{w} \in V$ by consistent completeness (of the definition of discreteness), since $\underline{u}, \underline{x} \leq \underline{v}$.

Now, $\underline{u} \leq glb(\underline{u},\underline{x}) \leq \underline{v}$, which can be written as $\underline{u} \leq \underline{w} \leq \underline{v}$, and $\underline{w}(l) = \max(\underline{u}(l),\underline{x}(l)) = \underline{u}(l).x$ so that $\underline{u}(l) < \underline{w}(l) < \underline{v}(l)$. This implies that $\underline{u} < \underline{w} < \underline{v}$, and we have a contradiction since $\underline{u} \lhd \underline{v}$. □

A corollary of Proposition 3 can give a formal description of the way transaction vectors are actually obtained.

Corollary 1. Suppose that $V \subseteq V_T$ is a normal transaction language and $\underline{u}, \underline{v} \in V$ such that $\underline{u} \leq \underline{v}$ (and also $\underline{u} \neq \underline{v}$), then there exists $\underline{a}_1, \ldots, \underline{a}_n \in A_T$ such that

1. $\underline{u}.\underline{a}_1 \ldots \underline{a}_n = \underline{v}$
2. $\underline{u}.\underline{a}_1 \ldots \underline{a}_i \in V$, i = 1..n
3. $\underline{u} \lhd \underline{u}.\underline{a}_1$ and $\underline{u}.\underline{a}_1 \ldots \underline{a}_{i-1} \lhd \underline{u}.\underline{a}_1 \ldots \underline{a}_i$, i = 2..n.

Proof.

If $\underline{u} \lhd \underline{v}$, then the corollary holds with n = 1 and $\underline{a}_1 = \underline{v} / \underline{u}$, by Proposition 3.
Otherwise, there exists $\underline{w} \in V$ such that $\underline{u} \leq \underline{w} \lhd \underline{v}$, by Lemma 1 (in Section 2.3.3). By induction, there exists $\underline{a}_1, \ldots, \underline{a}_{n-1} \in A_T$ such that

$$\underline{u}.\underline{a}_1, \ldots \underline{a}_{n-1} = \underline{w}.\underline{a}_1, \ldots \underline{a}_i \in V,\ i = 1,.. n-1,$$

and

$$\underline{u} \lhd \underline{u}.\underline{a}_1, \text{ and } \underline{u}.\underline{a}_1, \ldots \underline{a}_{i-1} \lhd \underline{u}.\underline{u}.\underline{a}_1, \ldots \underline{a}_i,\ i = 2, \ldots, n-1$$

If we now let $\underline{a}_n = \underline{v} / \underline{w}$, then by Proposition 3, $\underline{a}_n \in A_T$, $\underline{u}.\underline{a}_1 \ldots \underline{a}_n \in V$ and $\underline{u}.\underline{a}_1 \ldots \underline{a}_{n-1} \lhd \underline{u}.\underline{a}_1 \ldots \underline{a}_n$. This means that $\underline{a}_1, \ldots, \underline{a}_n$ have the desired properties. □

We may now formally define transaction vectors as series of concatenations with column vectors.

Definition 9. (***t*-decompositions**) Suppose that $\underline{u}, \underline{v} \in V \subseteq V_T$ such that $\underline{u} \leq \underline{v}$. We shall define sequences such as $\underline{a}_1 \ldots \underline{a}_n$ in Corollary 1 as *t-sequences* from $\underline{u}$ to $\underline{v}$. In the case that $\underline{u} = \underline{\Lambda}_T$, we describe such sequences as *t-decompositions* of $\underline{v}$.

Since the transaction vectors used in describing the behaviour of a long-running transaction are built up from the empty vector, they are *t*-decompositions. This means that the recursive application of the right-cancellation, on each action that has gone into obtaining the vector in question, cancels out the initial actions, leaving only vectors containing no observable actions as a result. In other words, it leaves the

corresponding transaction language only with the empty vector $\underline{\Lambda}_T$, effectively returning the system to (an approximation of) the state it was before the long-running transaction started.

It should be noted that in Deliverable D3.1 [RMK07] we have described an extended lock mechanism for recovery management and concurrency control. Thus, in our overall approach to long-running transactions in digital ecosystems we go beyond the assumption that a compensating action undoes the effects of its associated action. The corresponding locks, namely the internal lock (I_Lock), conditional-commit lock (C_Lock) and recovery lock (R_Lock), together with the corresponding IDG and EDG graphs, show how the dependencies between subtransactions due to data sharing are handled. In addition, the time-out lock (T_Lock) covers the cases of sequential alternative service composition and allows for forward recovery. Further details can be found in D3.1 [RMK07]. Such aspects have not been discussed in the corresponding formal setting, but work is in progress on their formal treatment.

2.3.8 Modelling forward and compensating behaviour of a transaction

In the previous section we described how the ordering relation between different vectors of a transaction reflects the orderings between activations of its sub-transactions. The vector-based description of behaviour in our formal model for long-running transactions makes it possible to express sequential, parallel and alternative behaviour of a transaction.

We have seen that the ordering relation between transaction vectors is given in terms of the coordinate-wise prefix ordering relation '$\leq$' of Definition 3. This turns the set of vectors associated with a transaction to a partially ordered set $(V, \leq)$ (recall Proposition 1). Since each vector describes the part of behaviour of the component in which the actions appearing in it have taken place, it is appropriate to say that whenever $\underline{u} \leq \underline{v}$, then $\underline{u}$ describes an earlier part of the behaviour described by $\underline{v}$. Further, in a normal transaction language, if $\underline{u} \lhd \underline{v}$ then the vector $\underline{v}$ describes the behaviour of the transaction in which a single action has (or concurrent actions have) occurred since $\underline{u}$.

Since $(V, \leq)$ is a partially ordered set some vectors may be incomparable. For example, consider the vectors $\underline{u} = (s_1, \Lambda, \Lambda)$ and $\underline{v} = (\Lambda, s_2, \Lambda)$ for which neither $\underline{u} \leq \underline{v}$ or $v \leq \underline{u}$. Such vectors describe either alternative behaviour (there is a choice between the last actions that went into forming each) or concurrent behaviour (the last actions that went into forming each are concurrent). Any pair of incomparable vectors stands in one relation or the other, and this is determined by what other vectors are in the set of vectors associated with a given transaction.

If the incomparable vectors are bounded above – in other words, if they describe earlier parts of some common later behaviour – then they describe concurrent behaviours. If they are not bounded above, then they describe alternative behaviours. It is important to stress that this is determined by context, by what other vectors are included in the set for a transaction.

This is illustrated in Fig. 2.20 which uses Hasse diagrams to depict the order structure of different sets of transaction vectors for a transaction with 3 leaves. It can be seen that s1 and s2 are sequential (s2 can only be activated after s1) in Fig. 2.20(i) while they are mutually exclusive (alternative) in Fig.2.20(ii) and they are concurrent in Fig. 2.20(iii).

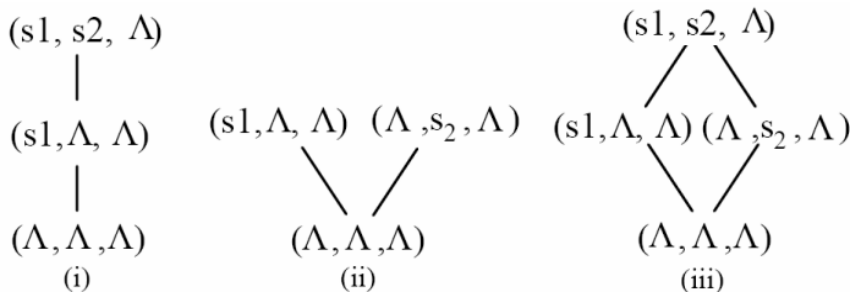


Figure 2.20 Order structure of transaction vectors

Notice that the set of vectors in case (i) does not include the vector (Λ, s_2, Λ). This, in addition with the fact that (s_1, s_2, Λ) is included, implies that s2 can only happen after s1 has (sequential dependency).

The set of vectors in case (ii) does not include (s_1, s_2, Λ). This has as a consequence that the vectors $\underline{u} = (s_1, \Lambda, \Lambda)$ and $\underline{v} = (\Lambda, s_2, \Lambda)$ are not bounded above in this case. Hence, the actions s1 and s2 are independent but do not take place consecutively in this case (one immediately after the other). This implies that there is a choice between doing s_1 and doing s_2 on the respective coordinates (alternative execution).

In case (iii) where the vector (s_1, s_2, Λ) is included, the vectors are bounded above and this implies that they describe the concurrent execution of actions s_1 and s_2 leading to the behaviour described by the vector (s_1, s_2, Λ). This is indicated by the familiar lozenge shape (or diamond) found in ATSs [Shi85], which marks the characteristic structure of a finite lattice [DaP90]. The incomparable vectors sitting at the middle of the lozenge are both available after the same behaviour (that is $(\Lambda, \Lambda, \Lambda)$ in this case) and occur consecutively leading to the behaviour described by the vector sitting at the bottom of the lozenge shape, i.e. (s_1, s_2, Λ).

Fig. 2.20 might be instructive with regard to the subtle distinction between independence and concurrency, which was discussed in Section 2.3.5. Independent actions are concurrent only if they are both offered after the same behaviour (are both enabled at the same point during the course of execution of a transaction). Otherwise, they may be mutually exclusive or even sequential.

It might also be worth pointing out that the lozenge shape in Fig. 2.20(iii) exhibits the characteristic structure of a finite lattice, which is a requirement of the discreteness property (Definition 4) in the case that the vectors $\underline{u}, \underline{v}$ are independent. The vector at the bottom of the lozenge is the least upper bound of the vectors in the middle, while the vector at the top is their greatest lower bound. This shows that discreteness – in the case of independent vectors bounded above in the set – is a property inherently related to concurrency.

The transaction tree shown in Fig. 2.14 has 6 leaves. The services s_1 and s_2 are to be executed in parallel (concurrently) followed by the data-oriented coordinator d_1. If the partial result released by d_1 (see Fig. 2.16) does not meet the desired outcome, then s_3 and s_4 are executed in succession (sequentially) followed by d_2.

To model the behaviour of the transaction in our formalism, we assign each leaf to a vector coordinate (from left to right here). This results in the set of 6-tuples shown in the Hasse diagrams of Fig. 2.21, which describe all possible series of subtransaction activations in performing the transaction T1 (given earlier in Fig. 2.14).

In Fig. 2.21 there is a choice between the behaviour described in the diagram on the left and that on the right, and this reflects the sequential alternative scenarios of transaction T1. This choice is deterministic and will be resolved on the basis of whether d_1 satisfies the desired outcome.

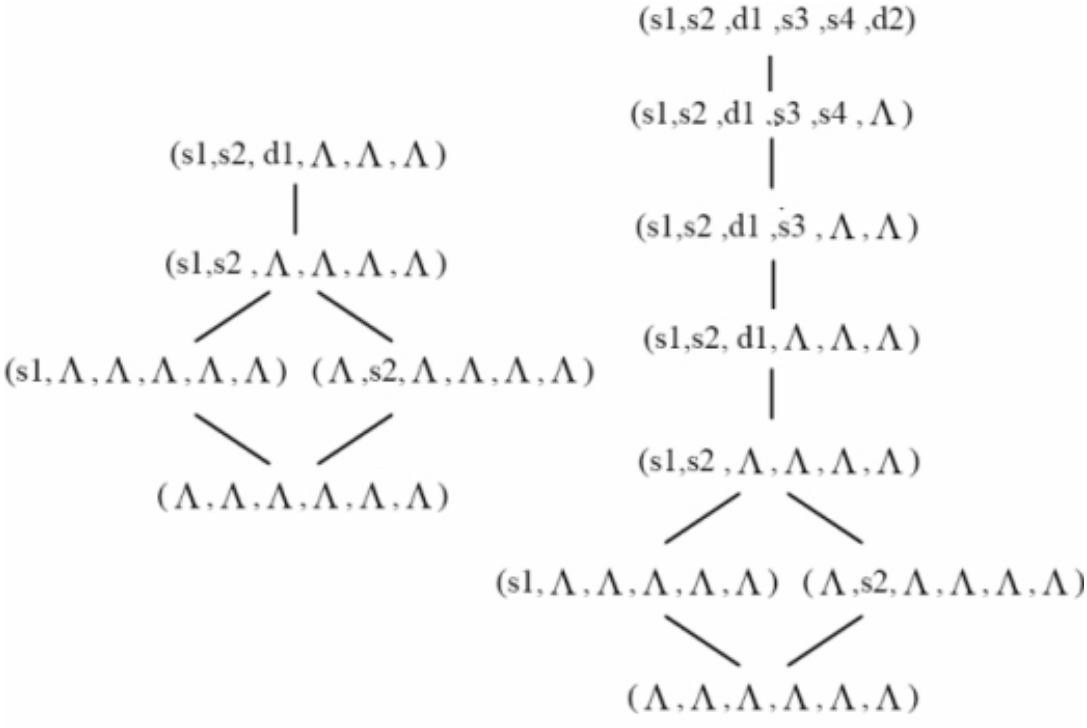


Figure 2.21 Transaction vectors for T1

Notice the lozenge formed by s_1 and s_2 which execute in parallel (in both cases). Also, notice that the Hasse diagram on the left implies that $(s_1, s_2, d_1, s_3, \Lambda, \Lambda) \leq (s_1, s_2, d_1, s_3, s_4, \Lambda)$ which means that s_4 can only happen after s_3 has (sequentially).

The Hasse diagram depicting the order structure of the transaction vectors for T1 can be readily used for checking against discreteness and local left-closure. For discreteness (recall Definition 4), we concentrate on vectors which have more than one vector immediately underneath. Then, that vector together with its immediate predecessors (the vectors immediately below it) must constitute a finite lattice. This will be the case when the immediate predecessors are bounded above (least upper bound) and below (greatest lower bound) by some vector in the set. In our example, such a vector is $(s_1, s_2, \Lambda, \Lambda, \Lambda, \Lambda)$ which has two distinct immediate predecessors, namely $(s_1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda)$ and $(\Lambda, s_2, \Lambda, \Lambda, \Lambda, \Lambda)$. These vectors are bounded above by $(s_1, s_2, \Lambda, \Lambda, \Lambda, \Lambda)$ and are bounded below by $(\Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda)$. Hence, the set V_{T1} is discrete.

For local left-closure (recall Definition 5), we look at each coordinate of the vectors. We concentrate on those which have a sequence of length greater than one. In such case, there must be some other vector in the set which, at the specific coordinate, has the same sequence but reduced by one. It can be readily checked diagrammatically that this is the case for the order structure depicted in Figure 2.21.

Furthermore, in case some subtransaction fails, the vector-based description is used in providing compensations for long-running transactions, taking up on the "do-compensate" and "validate-do" behaviour patterns. We have seen that compensations are performed by applying the right-cancellation operator '/' of Definition 3, which produces a unique vector, by Lemma 2. This unique vector is the immediate predecessor of the vector whose last action is compensated for. With successive applications of the right-cancellation operator we effectively move backwards along the Hasse diagram of Figure 2.21, removing the last action each time from each vector.

For example, consider the vector $\underline{v}$ = (s1, s2, d1, Λ, Λ, Λ). By applying Definition 9, with $\underline{u} = \underline{\Lambda}_T$ we have that the *t*-decomposition of $\underline{v}$ is $\underline{a}_1.\underline{a}_2.\underline{a}_3$, where $\underline{a}_1$ = (s1, Λ, Λ, Λ, Λ, Λ), $\underline{a}_2$ = (Λ, s2, Λ, Λ, Λ, Λ) and $\underline{a}_3$ = (Λ, Λ, d1, Λ, Λ, Λ). If a failure occurs after the action associated with doing d1, the actions s2 and s1 also have to be compensated since they are dependent on d1. This is done by applying the right-cancellation operator as follows.

First, we identify the last action that went into forming $\underline{v}$. This is done by looking for vectors in V which are immediate predecessors of $\underline{v}$. In this case it is vector $\underline{u}$ = (s1, s2, Λ, Λ, Λ, Λ) for which $\underline{u} \lhd \underline{v}$. By Proposition 2, we have that $\underline{v} / \underline{u}$ is a column vector – in this case it is $\underline{a}_3$ = (Λ, Λ, d1, Λ, Λ, Λ). Hence,

$$\underline{v} / (\underline{v} / \underline{u}) = \underline{v} / \underline{a}_3 = (s1, s2, d1, \Lambda, \Lambda, \Lambda) / (\Lambda, \Lambda, d1, \Lambda, \Lambda, \Lambda) = (s1, s2, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{u}$$

Similarly, by application of '/' on vector $\underline{u}$ we have,

Now, vector $\underline{u}$ = (s1, s2, Λ, Λ, Λ, Λ) is interesting in that it has two immediate predecessors, namely $\underline{x}$ = (s1, Λ, Λ, Λ, Λ, Λ) and $\underline{y}$ = (Λ, s2, Λ, Λ, Λ, Λ). Despite it being involved in a diamond the same process applies. In this case,

$$\underline{u} / \underline{x} = \underline{a}_2 = (\Lambda, s2, \Lambda, \Lambda, \Lambda, \Lambda) \text{ and } \underline{u} / \underline{y} = \underline{a}_1 = (s1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda)$$

Hence, we have

$$\underline{u} / (\underline{u} / \underline{x}) = \underline{u} / \underline{a}_2 = (s1, s2, \Lambda, \Lambda, \Lambda, \Lambda) / (\Lambda, s2, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{y}$$

$$\underline{u} / (\underline{u} / \underline{y}) = \underline{u} / \underline{a}_1 = (s1, s2, \Lambda, \Lambda, \Lambda, \Lambda) / (s1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{x}$$

Note we are now in the middle of the diamond appearing in the Hasse diagram of Figure 2.21. We apply similar development to both $\underline{x}$ and $\underline{y}$. The vector $\underline{x}$ has $\underline{\Lambda}_T$ as its immediate predecessor, and $\underline{x} / \underline{\Lambda}_T = \underline{a}_1$. So does $\underline{y}$, for which $\underline{y} / \underline{\Lambda}_T = \underline{a}_2$.

Hence, we have

$$\underline{x} / (\underline{x} / \underline{\Lambda}_T) = \underline{x} / \underline{a}_1 = (s1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) / (s1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) = (\Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{\Lambda}_T$$

and

$$\underline{y} / (\underline{y} / \underline{\Lambda}_T) = \underline{y} / \underline{a}_2 = (\Lambda, s2, \Lambda, \Lambda, \Lambda, \Lambda) / (\Lambda, s2, \Lambda, \Lambda, \Lambda, \Lambda) = (\Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{\Lambda}_T$$

In this way all actions that had occurred before failure, as described in vector $\underline{v}$ = (s1, s2, d1, Λ, Λ, Λ), have been compensated for. It can be seen that there are no longer any visible observable actions, as illustrated by $\underline{\Lambda}_T$.

It might be instructive to compare with the way compensations are handled in the approaches discussed in Section 2.2. In our approach there is no need for enforcing the sequence of actions to be performed in the reverse order. This is inherent to the way operations are defined on transaction vectors (right-cancellation, concatenation, ordering relations). Further, and perhaps even more importantly, no additional notation or formal construction is required in handling compensations for concurrent actions. Both compensating CSP [BHF05] and the approach taken by [BMM05], require additional syntax and a separate semantics in order to perform compensations for sequential actions that are composed in parallel. Finally, note that there is no need to consider different sequences of actions within a transaction and compose them in order to model concurrency in our approach. Concurrency is handled in terms of actions themselves, and there is no need for all actions within a transaction to be independent. This is one of the benefits of opting for a non-interleaving semantics, advocated by Shields [Shi85, Shi97] and Mazurkiewicz [Maz88], as it allows modelling true-concurrency between actions in a long-running transaction.

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 in terms of actions on the leaves (e.g. service invocations) and the coordination between them. The resulting behavioural patterns (see Fig. 2.21) 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 [Mos05]. We have also addressed compensations in an intuitive and relatively straightforward manner which does not require further formal constructions – it is again based on (coordinate-wise) operations on transaction vectors.

It might be worth pointing out that our formal description of the distributed transaction model here we have been concerned with modelling individual transactions, albeit in a way that allows to capture the release of partial results to other transactions. In other words, we have been mostly concerned with the dependencies within a transaction rather than between transactions. For the latter, it would appear that we need to consider the vectors from each and compose, in a principled way, in order to get the resulting inter-transaction behaviour. Previous work on composition within the vector-based representation of behaviour [MoS04] could be exploited in this respect.

Further, we note that the properties discussed in Section 2.3.3, discreteness and local left-closure, are shown to be preserved under composition of vectors in [MoS04]. Composition within this mathematical framework has also been studied in terms of the corresponding automata in [Shi05]. Finally, in [BoM06] we have considered a temporal logic interpreted over the vector language based description of behaviour we presented here, and in particular the automata generated by vector languages. As mentioned before, this is still work in progress and the automata generated by vectors as well as the extension with a temporal logic are currently under consideration.

3 Distributed Transaction Scenarios over a P2P Network

Conceptually we are dealing with a transaction vector world. The total composition of all of these vectors can create the business universe. Local Coordinators which are practical design for applying the transaction vector model may keep valuable information during the life time of a transaction. On the other hand, stability and predictability of a transaction depends on the pattern behaviour of each platform too (each SMEs). In this chapter, we are concerned with the coordination of open long-running transactions over a P2P network supporting a community of SMEs. In particular, we describe the use of a local agent that allows for the underlying services to be conducted in a truly distributed manner. At the heart of the local agent structure is the Local Coordinator which uses the information of other local agent components to orchestrate the necessary interactions involved in performing long-running transactions corresponding to complicated business activities. The way the local agent facilitates our transaction model is demonstrated by means of a simple scenario involving one service provider which has been used to guide the implementation of the basic concepts behind our transaction model.

3.1 Local Structure

Generally, web services do not need to make themselves known (in naming or any particular implementation) to the coordinator of a transaction. For this reason, we have designed an agent for each platform (SME) which control the communication and apply the vector transaction in one hand and keep the information about its local web services and external web services (belongs to other platforms) on the hand. Our live agent needs to have enough knowledge about its local web services to be able to deploy them based on our particular transaction protocol (reference to chapter 2 and D3.1).

On the other hand, promoting these web services can be one of the options which allows the agent give the possibility for other businesses to find out about this particular web service provider (business). Furthermore, other agents remotely can coordinate its platform - for example we can think about delegation in terms of avoiding heavy traffic.

At the same time, the agent can gather information about other web services not only for providing more possibility for its business but also for providing facilities for delegation (in terms of remote coordination of other web services when the coordinator is not at the same platform of web services), for creating different types of service composition by using other web services, for reducing the complexity of the potential lookup algorithm used, for finding alternatives for each web service, and finally for extending or optimising the corresponding business model.

Figure 3.1, shows an overview of the local agent structure. Such a system includes: a Local Web Services Informer, a Local Service Repository, a Web Service Information Investor, a Global Service Repository, a Web Services Promoter and a Local Coordinator. We describe each entity in more detail in the following sections (Figure 3.10 gives the full overview of the environment).

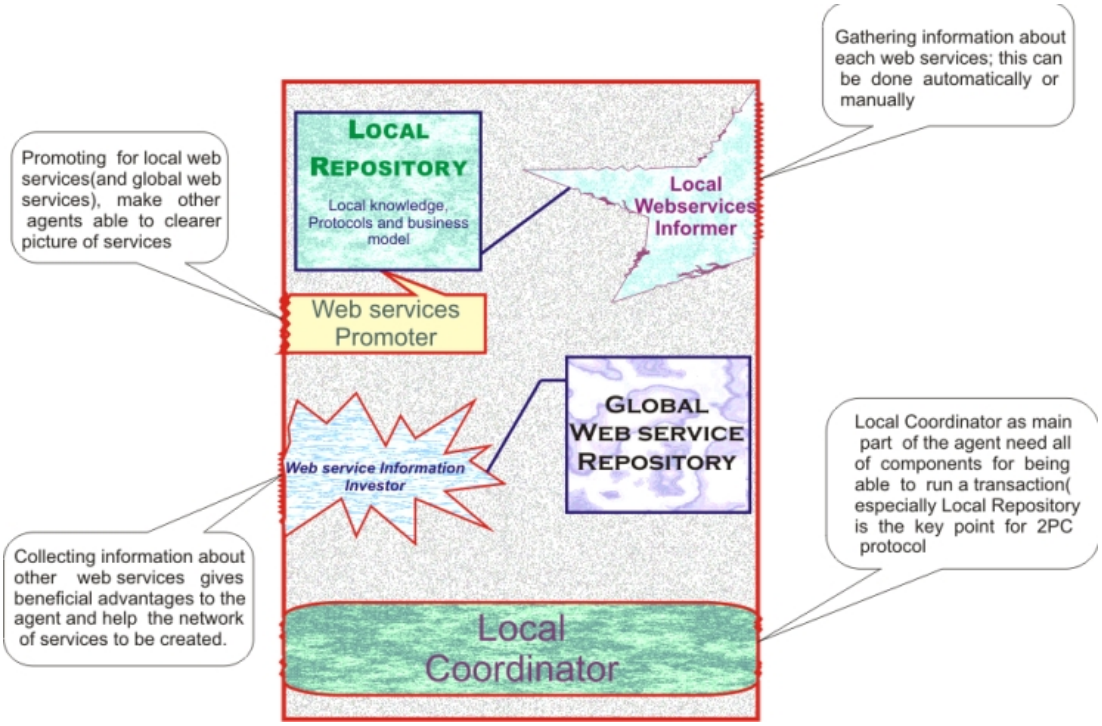


Figure 3.1: Overview of a local agent

3.1.1 Local Web Services Informer

Digital Ecosystems, at the centre of research work in OPAALS, are inherently operated in a heterogeneous environment which is service-oriented. On one hand this environment is supposed to support any type of web service, with any protocol in a loosely-coupled manner (local autonomy for SMEs) and on the other hand, it should support a proper commit protocol for long-running transactions (business activities as discussed in D3.1 [RMK07]).

In order to provide the Initiator of a transaction with the basic view about the web service which is to be to be used on its transaction, as well as the limitations / restrictions of the particular web service, we need to gather information about the web service. This information can be provided by each web service after its creation in some description language such as WSDL and/or can be provided manually by the SME which provides the web service.

Furthermore SMEs may change their web service protocol, parameters, etc., regularly and therefore the possibility for updating this information is necessary too. As a result, we need to provide two interfaces for keeping this information in the local agent as the component also requires an interface to the local repository (Figure 3.2 shows this component of our local agent).

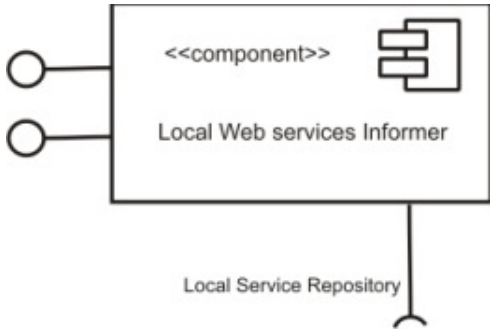


Figure 3.2. Local Web Services Informer

3.1.2 Local Service Repository

The Local Service Repository keeps information about each local web services in the platform (SME). This information is some description of each web service (for example it can be a SDL or WSDL) and any extra information such as availability, last updates and etc (which may help other SME to have clearer picture of that particular web service), can be included too. In the first place (as a component-based approach), the Local Service Repository should provide an interface to the Local Web Services Informer, e.g. for accessing the local web service description records. The next interface provided by the Local Service Repository gives access to the Local Coordinator to use the web service descriptions for creating and running a transaction.

Any updates, modifications or even the creation of web services should be promoted (at least for other partners with whom they are collaborating in running a transaction). That's the main reason the Local Service Repository requires an interface to another component, namely the Web Service Promoter, whose purpose is to promote the web services to other agents (Figure 3.3).

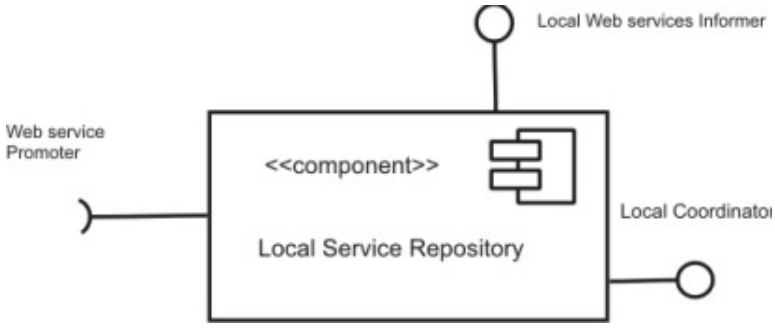


Figure 3.3. Local Service Repository

3.1.3 Web Service Information Investor

The structure of the local agent we considered in Figure 3.1 looks symmetric for both the local and the global view of web services. Therefore the Web Service Information Investor, as a symmetric component for Web Service Informer, does a similar job but this time for global web services.

It provides two interfaces for creating a new web services record and updating the current web services. Meanwhile, it requires an interface to the Global Service Repository (the symmetric component for Local Service Repository (Figure 3.4).

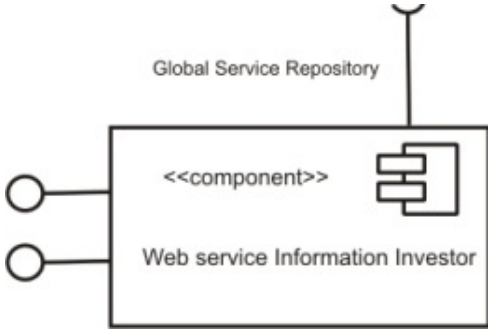


Figure 3.4. Web Service Information Investor

3.1.4 Global Service Repository

Similar to the Local Service Repository, the Global Service Repository provides two interfaces: one for the Local Coordinator to access the web services record descriptions and the other one for the web services Investor to make changes on the Global Service Repository.

The first interface plays a critical role for the Local Coordinator in making the decision about the protocol and the method for applying it on the transaction model. At the other side of the local agent is another SME which may change its web services descriptions regularly and even service availability can be an issue too. The second interface is important too, as updating the Global Service Repository is crucial.

On the other hand, the Global Service Repository should be able to inform the Web service promoter, as soon as any changes occur for its records. That is why it requires an interface to the Web service Promoter for doing that. Figure 3.5 shows this component.

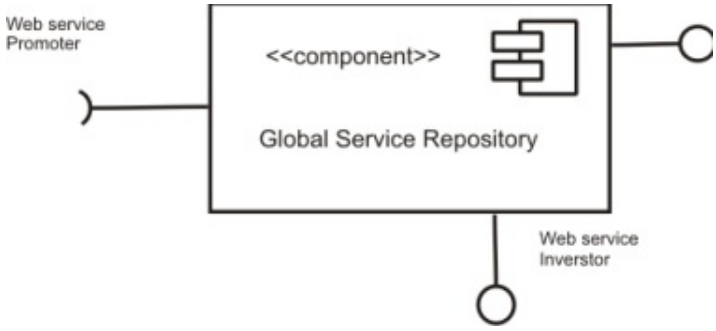


Figure 3.5. Global Service Repository

3.1.5 Web Services Promoter

The Web service Promoter is an important part of the local agent, as it reflects the situation of the web services of a local agent and the web services of any other connected agents to that particular agent. This can be done by using two interfaces which are provided for Local Service Repository and Global Service

Repository respectively. Meanwhile the Web Service Promoter requires two interfaces from the other agent to be informed of the latest situation of its local web services and any other web services which are communicating with it (Figure 3.6).

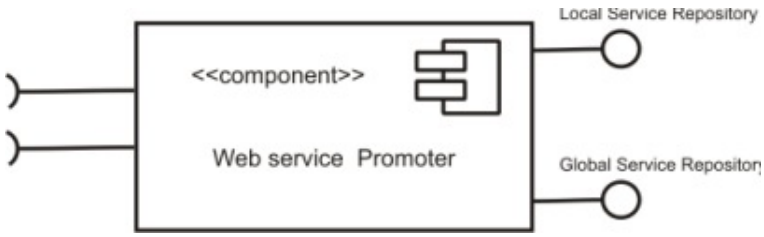


Figure 3.6. Web Service Promoter

In fact two interfaces for this component should be provided by the Web Service Information Investor of the other agent. Figure 3.7 shows this connection between Local agent A and Local agent B. When any changes happen for some records of the local or the Global Service Repository, they use the Web service Promoter's interfaces. The Web service Promoter in turn can use the interfaces provided by the Web Service Information Investor in Local agent B, and the Web Service Information Investor at agent B can update its Global Service Repository if needed (because in some cases it could be done already). As a result, the Global Service Repository of agent B will use the same interfaces for the Web service Promoter at agent B and this will be done for any connected agent to Local agent B. In this way, any changes on connected agents can be updated quickly.

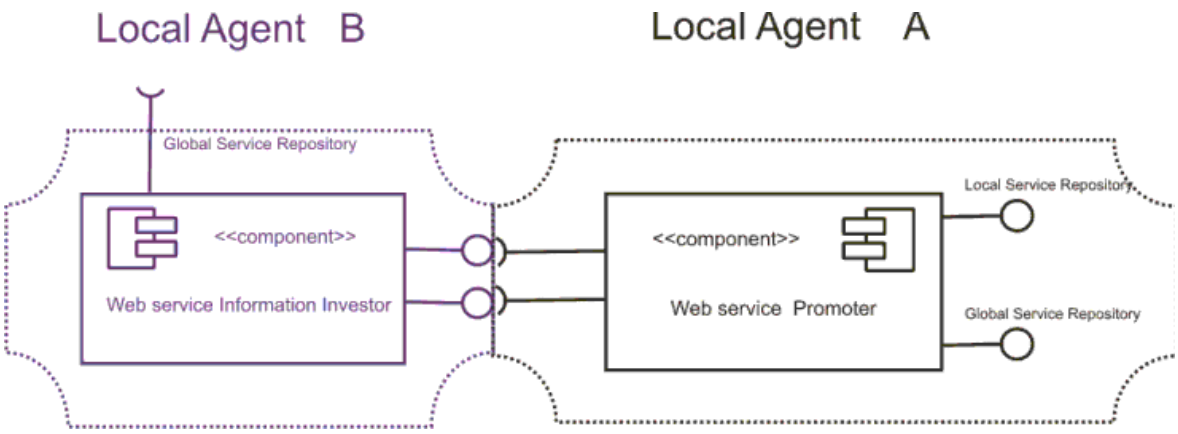


Figure 3.7. Web service Promoter role between two different agents

3.1.6 Local Coordinator

The kernel of the local agent is the Local Coordinator. Other components provide information for a Local Coordinator (on the local machine or even for a remote agent). The Local Coordinator facilitates our transaction model to be applied for complicated business activities (long-running transactions) as well as simple transactions.

Generally the Local Coordinator requires an interface from the Local Service Repository for gathering the information about local web services which enables it to provide the preparation and commit phase in a two phases commit (2PC) protocol. This normally can be handled by a transaction context in response to a transaction request (Script). Later in this chapter we will cover this by example.

For communicating with another agent (its Local Coordinator), the Local Coordinator as well as providing an interface, requires an interface from the remote agent too. The Local Coordinator also requires an interface from the Global Service Repository, especially when it acts as an Initiator of the transaction. This makes it possible to create the transaction script based on the knowledge of the other agents' web services. Ultimately, it requires the interface from its local web services to able to invoke them (see Figure 3.8).

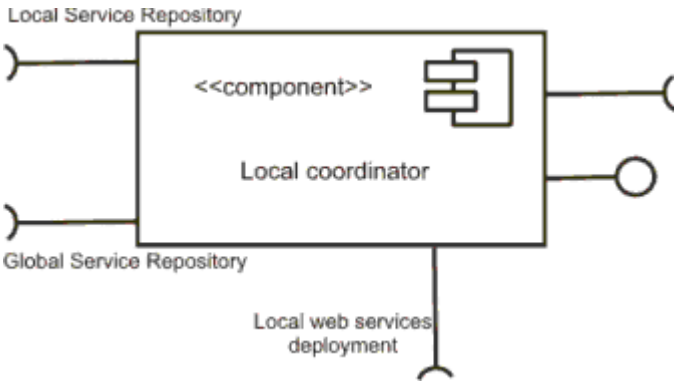


Figure 3.8. Local Coordinator

3.1.7 Local agent components and interactions

Figure 3.9 shows all components of a local agent (we consider this schema for any service provider in the system). The local agent by using two repositories (Local and Global service repositories) tries to provide detailed information about local web services, but also general information about other web services. This enables the Local Coordinator to invoke its local web services based on different protocols (for example 2PC or 3PC) and on the other hand, by using general information about other (remote) web services, in some sort of XML description, such as WSDL or SDL[5], it can create the transaction context (requirement).

The Local Service Repository should be updated by the Local Web Services Informer (any changes or updates can be effected on the Local Service Repository). Meanwhile the Local Service Repository can promote its services to other agents through Web Service Promoter.

The Global Service Repository can be updated by the Web Service Information Investor and at the same time, can promote these web services (which are stored in Global Service Repository) to the other agents (any changes will be promoted too, and in this way other agents can update their Global Service Repository).

[5] SDL (Service Description Language), it is a standard description language which is introduced in DBE project and it is popular in Digital Ecosystem community.

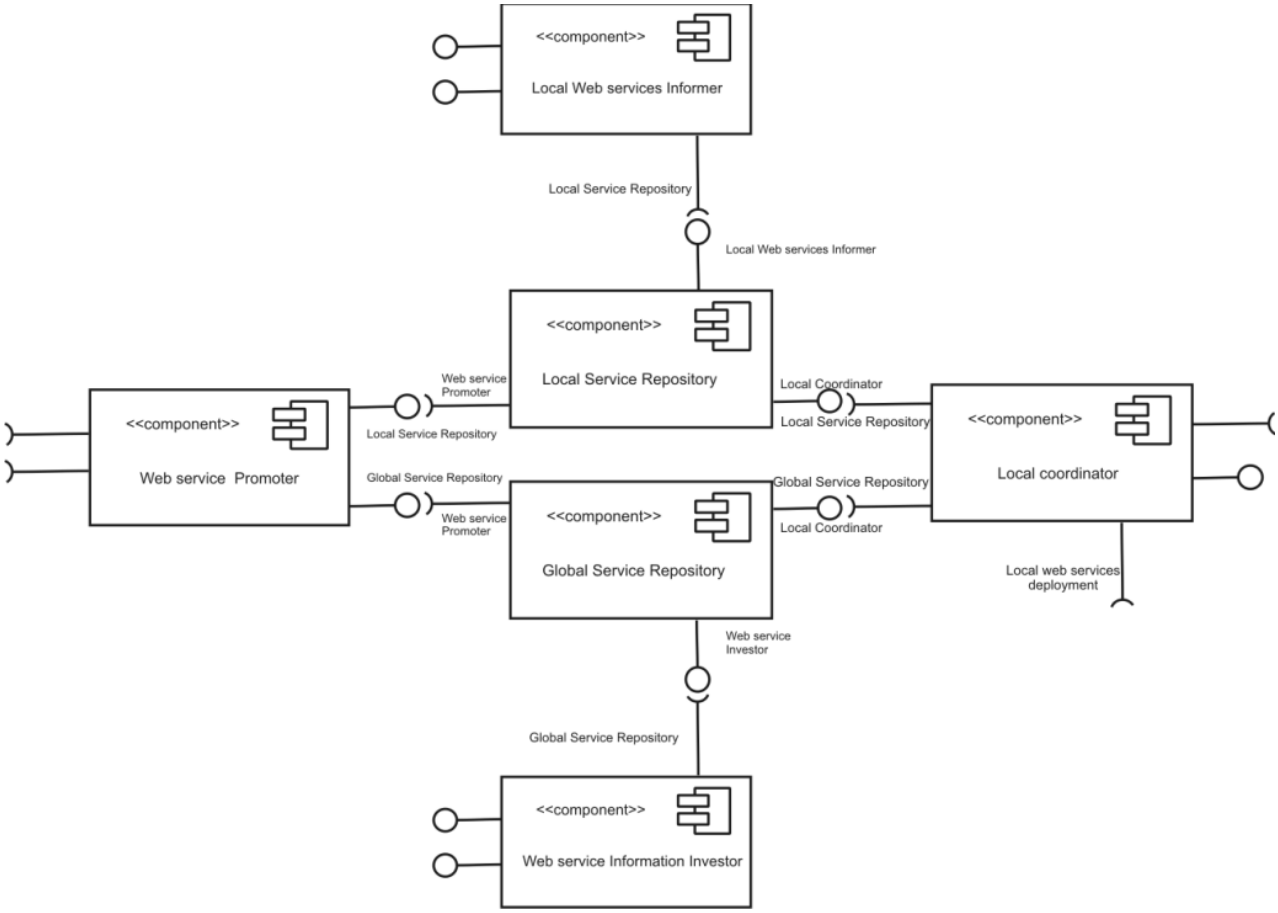


Figure 3.9. Local agent components

Figure 3.10 shows how communications between components of agents can improve the performance, can keep all agents' repositories updated and can provide enough information for the Local Coordinator of agents to run transactions.

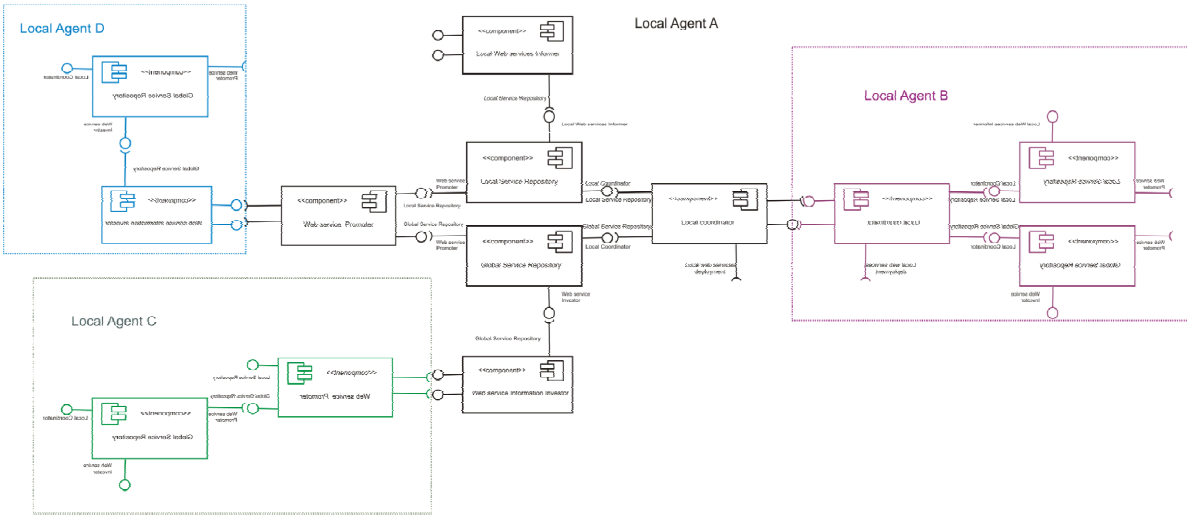


Figure 3.10. Overview of Multi-agent environment

3.1.8Towards Implementation

As the current crisis on experiencing serious performance deficiency from having to wait for a server's response during web service invocations (over the network such calls can take unpredictable lengths of time), JAX-WS 2.0 provides a new asynchronous client API, which can be applied for many clients, especially interactive ones such as JFC/Swing-based desktop applications.

By using this API, programmers are able to trust in the JAX-WS runtime to manage long-running remote invocations for them (they do not need to build threads on their own). In this way, asynchronous methods may be used in conjunction with any WSDL-generated interfaces as well as with the more dynamic Dispatch API. Sun Microsystems proposes two usage models:

- Polling model: you make a call. When you're ready, you request the results.
- Callback model: you register a handler. As soon as the response arrives, you are notified.

Note that when a WSDL document is imported, asynchronous methods are required to be generated for any of the operations defined in the web service. Furthermore, asynchronous invocation support is entirely implemented on the client side, so no changes are required to the target web service and there is no violation of service-oriented architecture or the SMEs' local autonomy. In the next section, we propose the practical scenarios which can use both usage models in practice. Further to this, as we rely on software agent on our design in contrast to the conventional application, abstractly the life cycle of this agent is not limited and later on (next deliverable), we add mobile agents for additional performance and flexibility to our design.

3.2 Local coordination based scenarios

We can start with a very simple example. In collaboration with TechIDEAS we started with a taxi reservation example which involves a simple taxi booking service. Firstly, we cover B2B transactions. Secondly, we try on one hand to improve the performance of the well-known transaction models WS-Trans and BTP by applying the new concepts of our transaction model, and on the other hand we consider where we are compatible with them.

This example may not reflect the whole spirit of the architecture as it is mostly aimed at clarifying the 2PC (two phases commit) protocol and the structure of the messaging between the service provider agent and the initiator of the transaction. We stress that in our model each of them has a separate coordinator and are loose-coupled so that local autonomy is not violated. Furthermore, the dynamicity of the environment and the potential requirement for applying more complicated scenarios is considered (we describe a more complicated scenario following this example).

3.2.1One service provider and one service scenario:

An initiator (customer), needs to reserve a taxi for a specific time, for a specific destination and through a specific taxi service company. The taxi service has a web service for responding to this request, which firstly does the booking and shows availability of a taxi for that specific time and then finalises the booking and also charges the customer. The sequence diagram given in Figure 3.11 shows the necessary interactions in this simple scenario.

Expected transaction response time

The important parameter here is the ‘Expected transaction response time’, which shows the initiator has considered a maximum response time for his/her request (transaction).

Because of the dynamicity of the environment and even the nature of transactions (a transaction after passing the specific time period will lose its reason for being executed), this parameter is important and can be set globally based on the latency of the environment, but it should be possible for being overridden by any initiator based on the nature of its request. Normally, the default version for conventional (database) transactions is for this parameter to be set globally, but for long-term transactions this parameter should be overridden (means, expected transaction response time can be differ from a transaction to the other) and this applies to real-time transactions too. If the expected transaction response time is passed, the transaction is supposed to abort automatically.

Two phases commit

Conventionally a two-phase commit (2PC) protocol is provided for covering distributed and advanced transactions. The two-phase commit protocol is a distributed algorithm that lets all nodes in a distributed system agree that a transaction can commit. The protocol results in either all nodes committing the transaction or aborting, even in the case of network failures or node failures. However, in terms of the classic 2PC, the protocol will not handle more than one random site failure at a time. The two phases of the algorithm are the preparation phase (commit-request phase), in which the coordinator attempts to prepare all the cohorts (in the simple way, these can be understood as web services), and the commit phase, in which the coordinator completes the transactions at all cohorts.

In our model, firstly we consider an extended version of 2PC in which we can cope with several failures (theoretically n–1 failures, where n is the number of non-critical subtransactions). In addition, we have introduced a new version of three phases commit in deliverable D3.1 for specific areas in which business activities need this expansion. In the next step we try to show the details of this example.

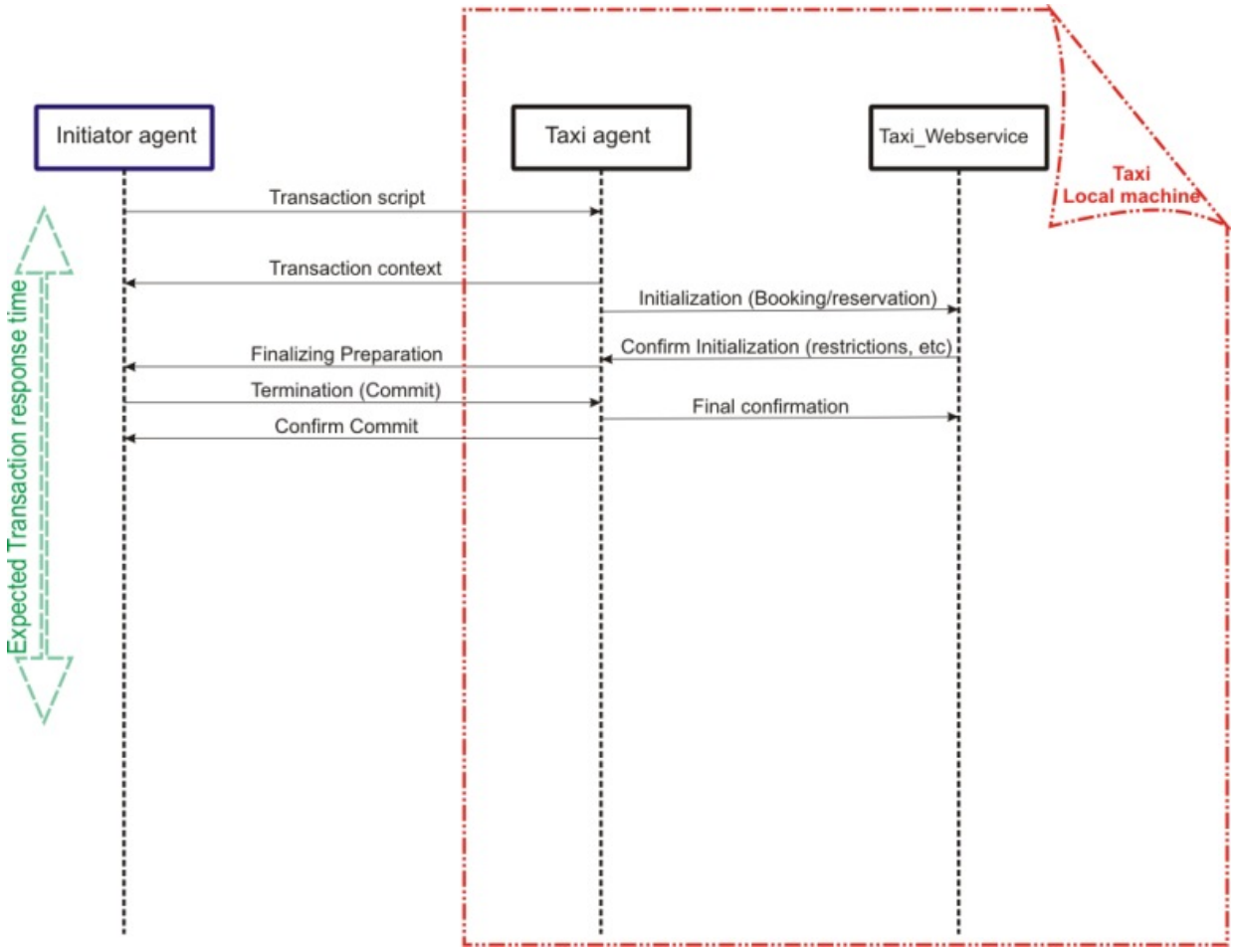


Figure 3.11: Simple-commit

3.2.2Transaction Script:

The initiator makes his request by using an XML message which is called *transaction script*. In this message the initiator determines the 'Expected Transaction Response time' (ETR) and the details of the required web services, the order of invocation and their alternatives, plus all necessities of the transaction for clarifying the customer's requirements. Also, the transaction protocol should be determined - necessary services which must be called, their location, id, parameters, the relationship between them, etc.

3.2.3Transaction context:

The Transaction context is a message which sets the road map for deploying the transaction and clarifies the limitations and restrictions of each web service, in terms of determining the protocol for communication, termination conditions, right for commit, service id, etc.

As an example, the PTE (Preparation Time-out Exception) and other parameters should be set in this context which can change the behaviour of all participants (any coordinator/agent in the transaction including the initiator). This parameter will be clearer during this document and its application in new/exceptional circumstances will be described.

3.2.4 First phase of 2PC:

The first step includes opening the session by sending the 'Transaction script' and responding with the 'Transaction context'. In this step the transaction requirements can be expanded, in terms of the methods and limitations (for example, by determining PTE, the initiator will realise, if it does not respond with a confirmation for initialisation within a specific time, that abortion will happen - the initialisation will be cancelled based on the taxi agent's decision; if there is still time, based on ETR, it can try the 1st phase again).

Initialization

Based on the transaction description, the taxi agent (which actually is a coordinator), tries to do a primary booking on one of its taxis by calling the Taxi_Webservice.

Confirm initialization

When Taxi_Webservice has done the primary booking, it can send a confirmation message which can include the id of that specific taxi, the time period for reaching the destination and other specification. Meanwhile other restrictions for a primary booking may apply; for example the web service does a primary booking for taxi with id 12 for 20 seconds (PTE) and if it does not finalise within the next 20 seconds it will cancel the booking (and in exceptional way such as Bob's Taxi service it may do an automated confirmation, depending on the corresponding business model).

Finalizing preparation

When the taxi agent (its Local Coordinator) receives confirmation for the initialisation, it will send a 'finalizing preparation' message which shows the details of the primary booking (Taxi id and/or any other necessary information) and its limitations (such as the time for validation of that, i.e. PTE). At the same time it is supposed to be the last message of preparation phase.

3.2.5Second phase of 2PC:

The second phase of the transaction can be started after the 'Finalizing Preparation' message. Based on the 'Transaction Script' and the 'Transaction Context', the right for starting the second phase can be clarified. In this example, like most cases, this right is with the initiator and it is up to the response of the initiator to the last message of the Taxi agent. In the other cases, entering the second phase can be done after passing the corresponding PTE set, i.e. not waiting for the response from the initiator.

Termination:

The first message for entering the second phase is 'Termination' (with commit signature) which announces to the recipients (here Taxi Agent) that they may proceed to finalise the primary booking.

Final confirmation:

As a response to the termination message, each service provider tries to finalise its services. In our example the Taxi agent sends a final confirmation (can be done by calling the Taxi_Webservice with an appropriate parameter for confirming the booking and debiting the associated cost from the customer's account).

Confirm commit:

The 'Confirm Commit' is a receipt which confirms the transaction commit. It is more like sending the invoice to the customer after the job is done and the cost is debited.

3.2.6 Latest commit:

Figure 3.12 illustrates the latest possible successful commit and it clarifies that ETR is the expected time for receiving the 'Finalizing Preparation' message. In addition, ETR is the expectation for entering the second phase – normally, the 1st phase occupies most of the time of a transaction and in the 2nd phase coordinators try to commit the transaction or cancel it based on protocol.

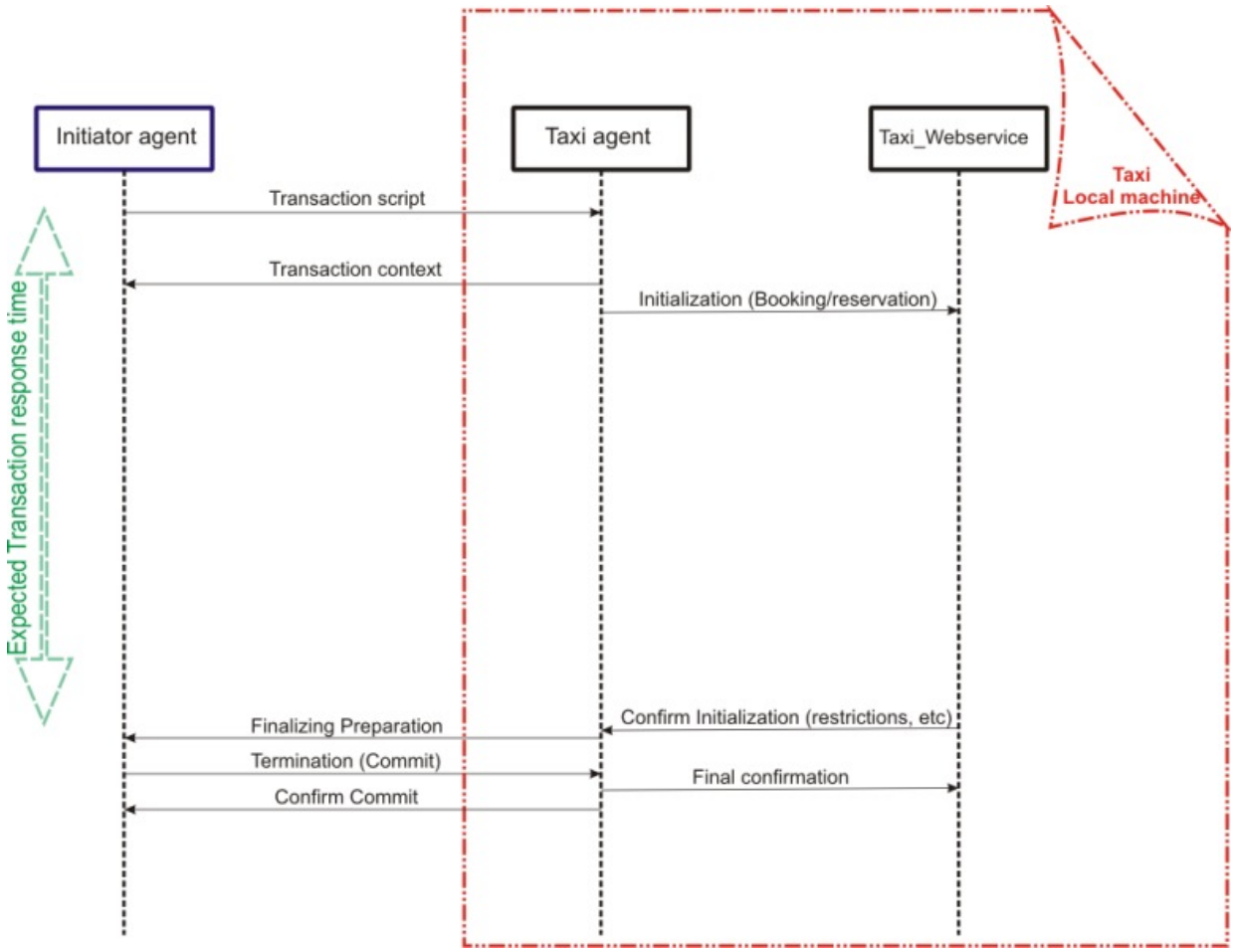


Figure 3.12: Latest commit

In this example, you can imagine a situation where a few taxies have been associated with a primary booking and they have not finalised the booking. That is why the web service does not respond immediately and waits for a specific time for them to finalise or release, and in this case one of them has been released. For example, it was an unsuccessful booking with another customer and it becomes available for booking by the initiator.

3.2.7Service unavailability (simple cancellation):

Figure 3.13 shows the standard situation of cancelling a transaction. In this example, after the Taxi agent calls 'Taxi_Webservice' the specific web service cannot find any taxi to book (for example all of them have been booked and their booking is finalised by other customers).

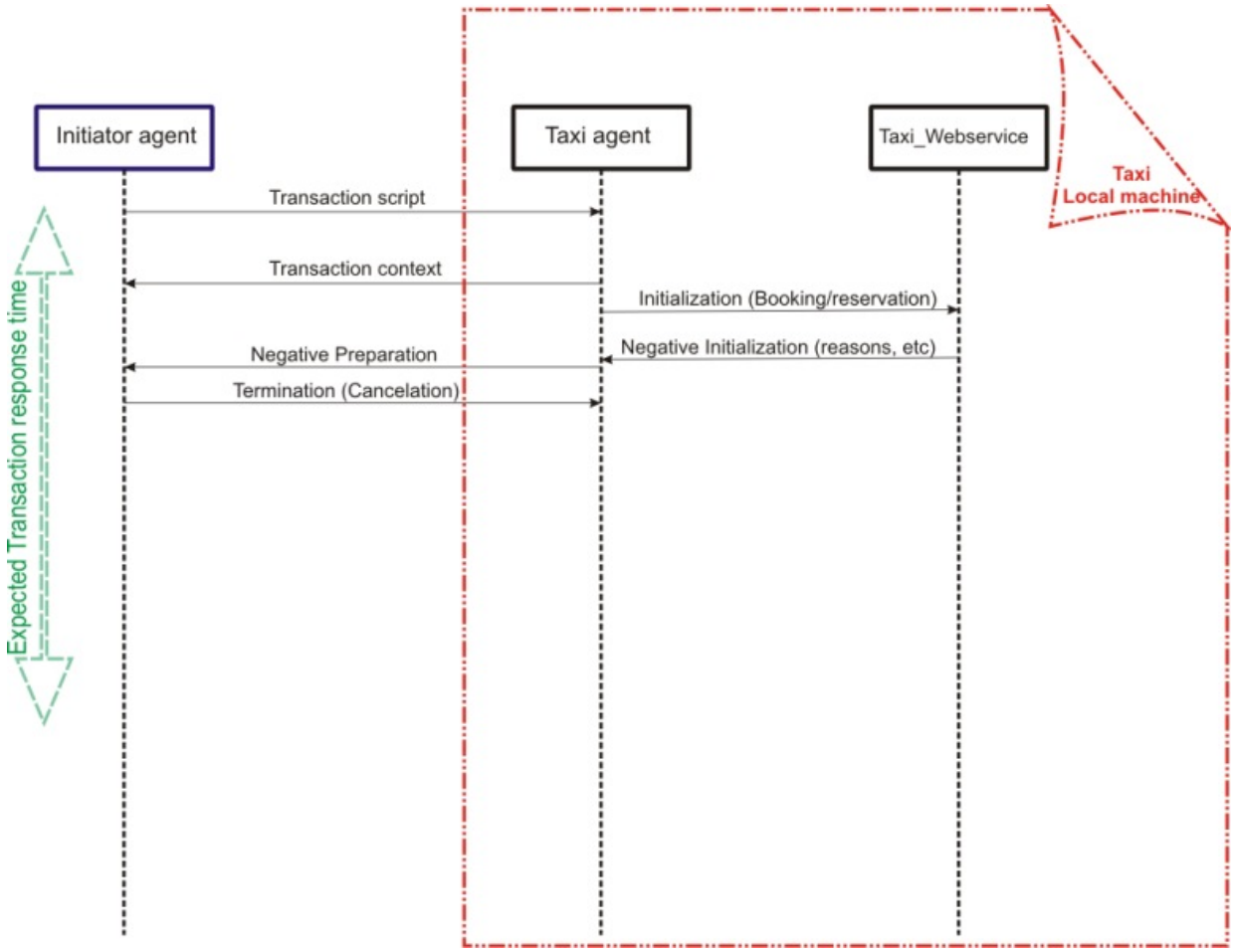


Figure 3.13: Simple cancel

Hence, its response is negative initialisation which means the transaction has to be cancelled and the termination message should be sent by a cancellation signature. Note that in this example the right for termination is with the initiator.

3.2.8Late Response (self-operation):

Figure 3.14 and 3.15 show the situation where there is too much delay by the service and it reaches the pre-set ETR. Naturally, the transaction cannot continue any more and has to be cancelled. This cancellation can be done by any party in the transaction (as all of them know the time limitation for the transaction life time).

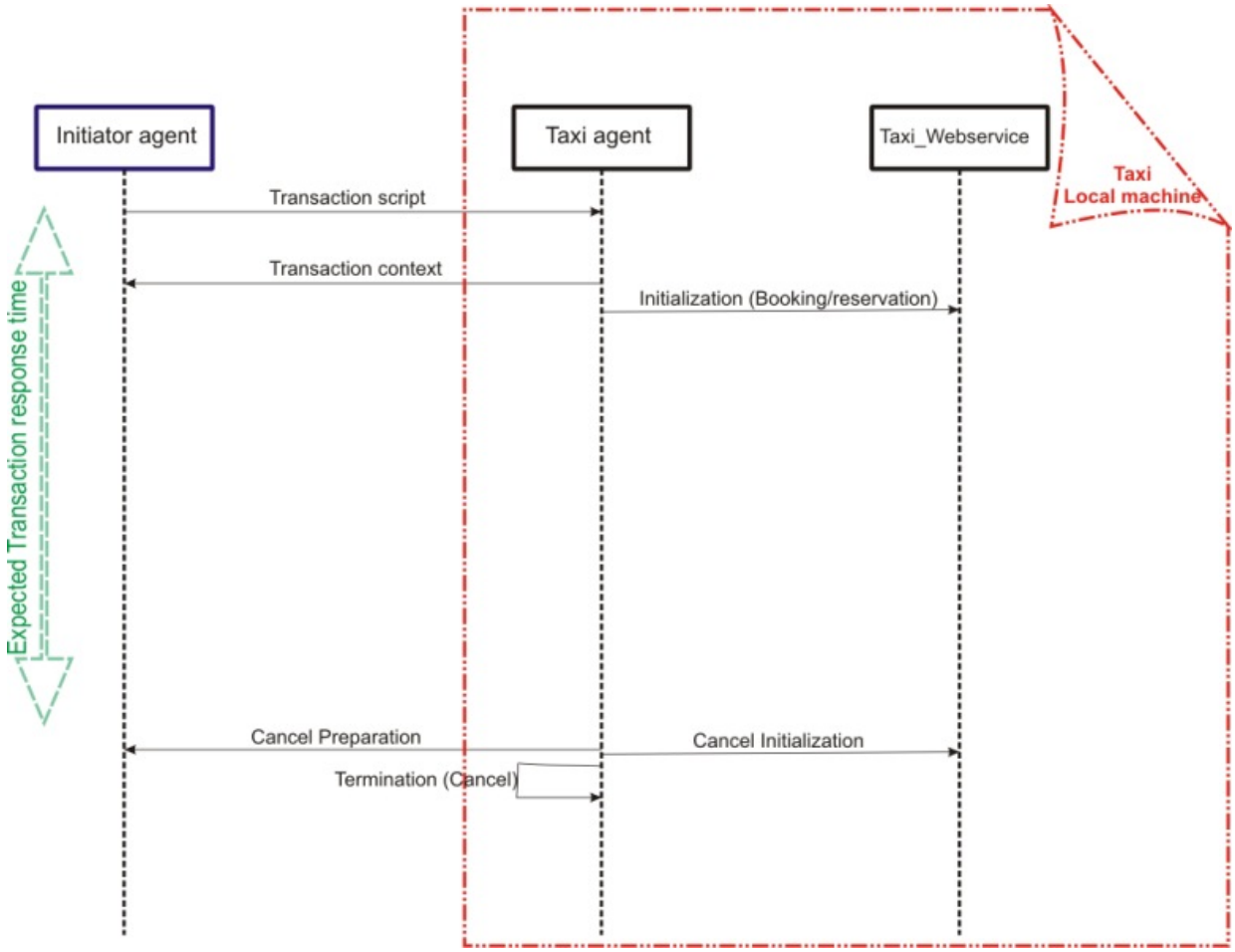


Figure 3.14: Late response

In Figure 3.14 the Taxi agent realises the fact the waiting time is passed and sends the cancellation to the web service and the initiator while in Figure 3.15 the initiator sends the cancellation (in the logic of cancellation that does not make any difference).

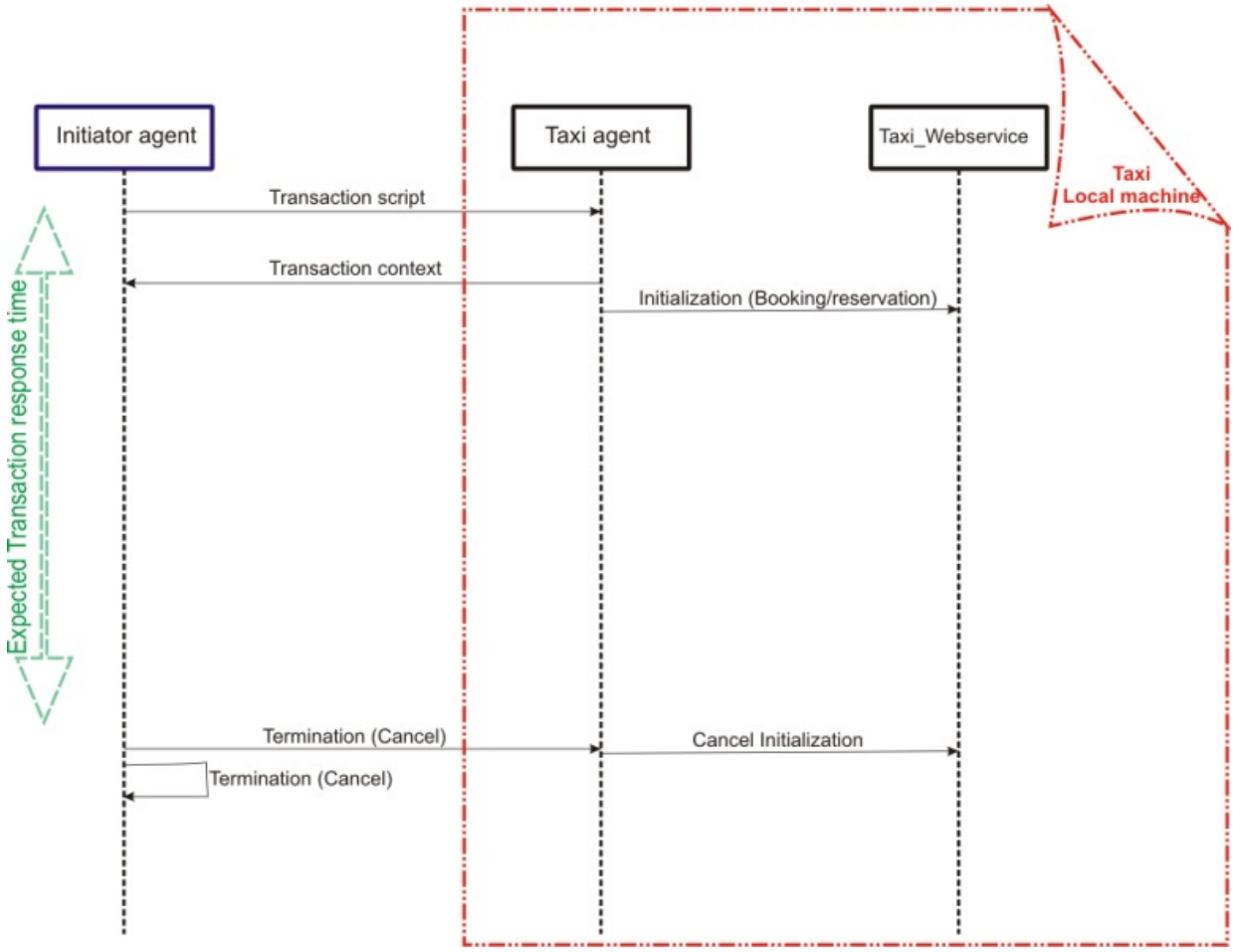


Figure 3.15: Late response (2nd case)

3.2.9Exception:

Based on the complication of the scenario, we can analyse different kinds of exceptions. One of the most important exceptions, which can even happen with only one service provider, is the Preparation Time-out Exception.

Preparation (phase) Time-out Exception

Preparation Time-out Exception (PTE) happens when the web service is waiting for 'Final confirmation'. Based on the nature of the business, the primary booking for a service (a taxi or a seat on the airplane) is limited to a pre-defined time-out by the web service provider. If it is passed, the primary booking will be cancelled automatically to allow other clients to book that specific service.

In our example, after the Taxi agent sends its response for finalising the 1st phase and start of the 2nd phase of the transaction, it does not receive any response from the initiator. After passing the PTE, it has to make a decision based on the transaction protocol used. Conventionally, the decision is cancelling all web services and waiting for the initiator to respond. If the response is before ETR, the Taxi agent can send the Transaction context for the initiator which means that the primary booking has to be done again and the transaction is still in the first phase.

As the initiator has the PTE, normally we do not expect the situation like Figure 3.16 to happen; therefore, the initiator may not send 'Termination(commit)' message for entering the 2nd phase and it may repeat the transaction script with reduction of ETR by the time which is passing. However, in occasional circumstances (such as restarting the initiator terminal and loosing some parts of the script, or simply network delays) such a situation may arise. The Taxi agent response clarifies whether this is the case. After that, if still there is time (based on ETR) the transaction will have a chance for successful commit.

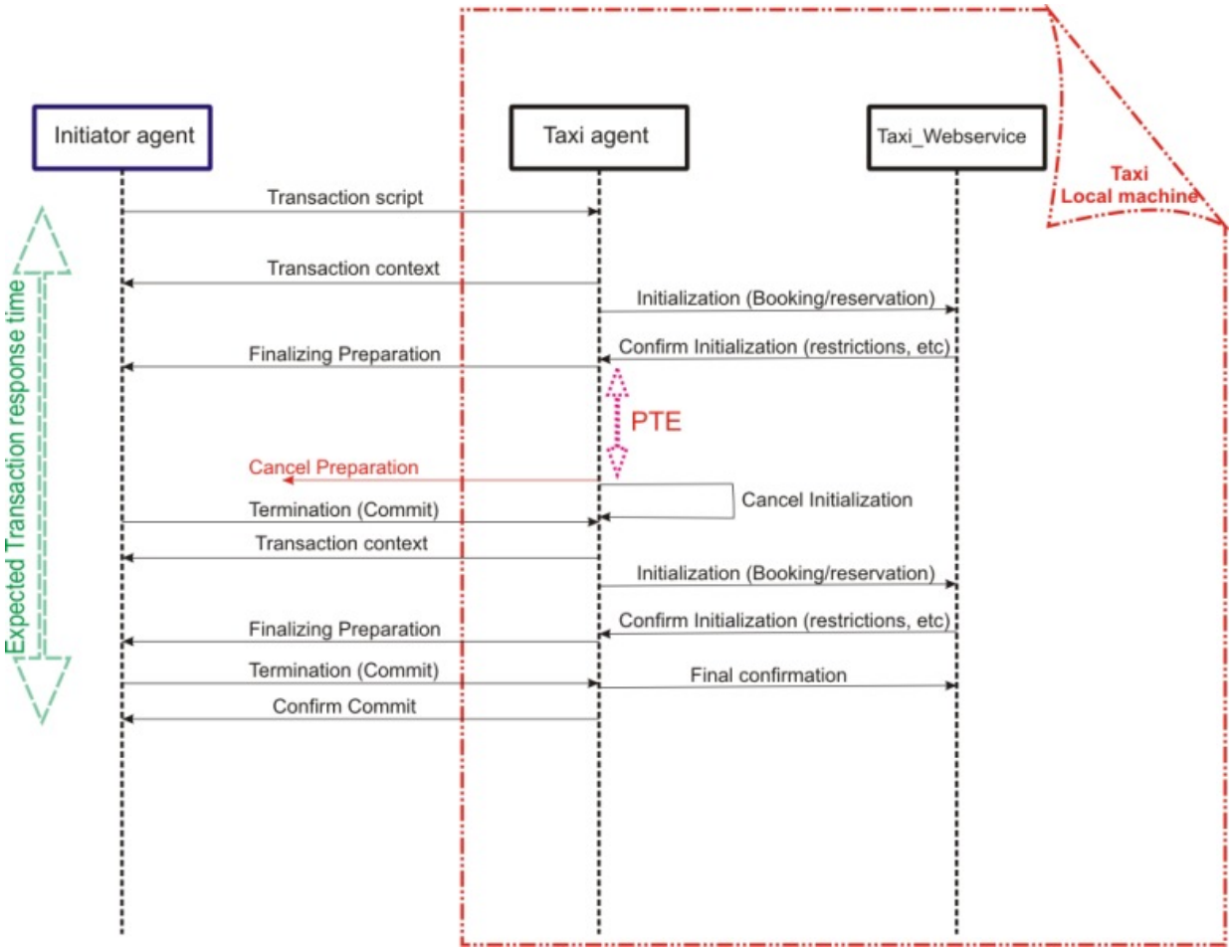


Figure 3.16: Preparation (phase) Time-out Exception

Another less popular protocol for processing a transaction in similar circumstances can be considered for giving different rights to the participants of the transaction. For example, the Taxi agent could have the right for starting the 2nd phase, which means after passing the PTE it could start the second phase of the transaction (as it did not receive any response from the initiator, it considers that as an agreement to the commit). Therefore, the Taxi agent could send the 'Final confirmation' to the Taxi_Webservice immediately after the PTE has passed and at the same time let the initiator know that (and/or waiting for initiator to be activated/online to receive the confirm commit as an invoice).

4 Analysis of Network Parameters: Experimentation and Simulation

In this chapter we begin to take closer look at the interrelation between distributed business transactions and the P2P network that supports them. The aim is to identify opportunities where the network parameters can be fine tuned to better facilitate the long-term business activities taking place between different parties, but also any points where the characteristics of the transaction model itself can be used to optimise the performance of the P2P network.

4.1 Business Transactions and P2P Network

Our primary model and analysis shows the specification of the network which supports open business transactions in a fully distributed manner and avoids violating the local autonomy of SMEs. As we analysed the specification of such a network in D3.1, our current model is trying to fulfil those requirements. From Section 4.1 to 4.4 we specify the network structure and in Section 4.5 we outline the evolutionary model which inspires this research.

A digital ecosystem involving open communities of SMEs is a highly dynamic environment and thus it is important to consider the way the network responds to changes in the number of nodes. We discuss an evolutionary framework, inspired by biology, and describe how it can be adapted and applied to the P2P network considered in WP3 in a way that respects its primary characteristics.

4.1.1 Current Business Transactions

For the simple transactions (adhering to ACID properties), the initiator of the transaction tries to control the workflow (Figure 4.1). A centralised control provides the atomicity, consistency and isolation of the transaction and durability of the results is supposed to be provided by the initiator (and sometimes other involved parties of a transaction). But in advanced applications these properties present unacceptable limitations and can reduce the performance dramatically [Elm94].

A high range of B2B transactions (business activities [CCC+05] or business transactions [RKM06]), has a long execution time period. Strictly adhering to ACID properties for such transactions can be highly problematic and can reduce concurrency dramatically. Further more, the lack of reliability and availability of the initiator, can bring more instability to the business environment.

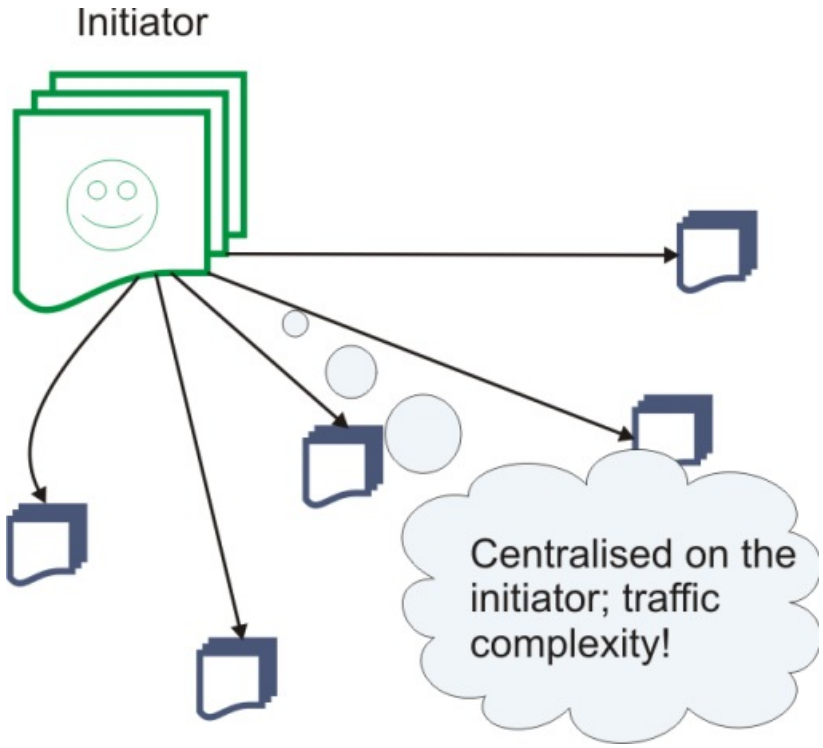


Figure 4.1. Traditional transaction processing by an initiator

In 2001, a consortium of companies including Oracle, Sun Microsystems, Choreology Ltd, Hewlett-Packard Co., IPNet, SeeBeyond Inc. Sybase, Interwoven Inc., Systinet and BEA System, began work on the Organization for Advance Structured Information Systems (OASIS) Business Transaction Protocol (BTP), which was aimed at business-to-business (B2B) transactions in loosely-coupled domains such as Web Services. By April 2002 it had reached the point of a committee specification (see [CDF+03] and [FDF+04]).

At the same time, others in the industry, including Microsoft, Hitachi, IBM, IONA, Arjuna Technologies and BEA Systems, released their own specifications: Web Services Coordination (WS-Coordination) and Web Services Transactions (WS-AtomicTransactions and WS-BusinessActivities) [CCJ+04]. Recently, Choreology Ltd. started an effort for a joint protocol which attempts to cover both models. A number of open problems with each protocol have been encountered and these are detailed in several reports by Choreology, e.g. see [FuG05].

The detailed analyses of these models have been discussed in deliverable D3.1 [RMK07] but in general they rely on a heavy coordinator which it is provided by a large enterprise and provides correctness and recoverability (Figure 4.2). The primary assumption of these two is tight coupling between participants and coordinator, violating the local autonomy of SMEs (because of violation of loose coupling). Furthermore lack of forward recovery and omitted results brings considerable negative impact on the SMEs. Currently, the centralised control of a transaction is provided by large enterprises (LEs) which is something that forces the involvement of large enterprises in the transaction and ignores the fully distributed aspect of digital ecosystems.

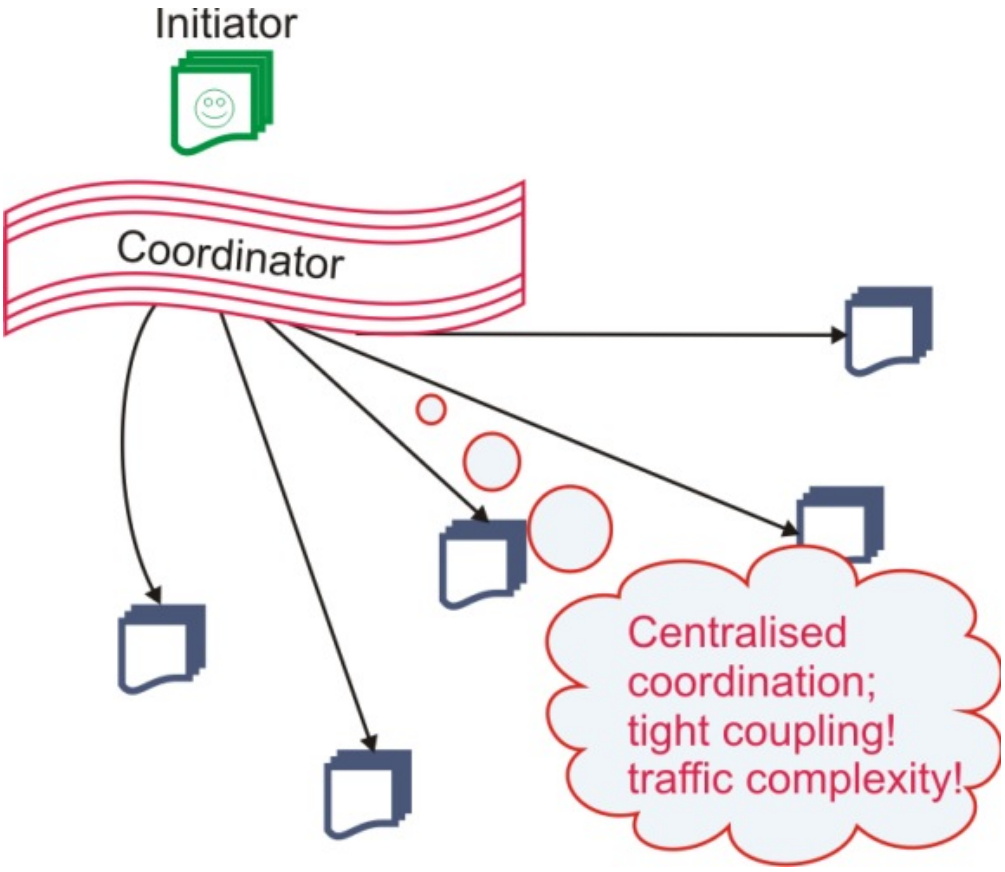


Figure 4.2. Applying coordinator framework (e.g. WS-Coordination)

4.1.2Distributed Transaction model (OPAALS/DBE)

In D3.1, we have described a distributed model of multi-service long-running transactions which has been designed for open collaborations within a community of SMEs in digital ecosystems. The model considers various forms of service composition to cover a wide spectrum of business scenarios. It allows the sharing of uncommitted results within a transaction as well as the exchange of results across transactions before their final commitment (partial results). At the heart of the model are the distributed log structures, provided by the IDG and EDG graphs, which in combination with the fine-grained lock mechanism allow for maximal concurrency in the transactional environment of a digital ecosystem.

The transaction model (in OPAALS/DBE) provides the capability for efficient recovery management, in terms of preserving as much progress-to-date as possible (omitted results), and provision for alternative scenarios or paths of execution (forward recovery). Using local coordination not only avoids any violation of local autonomy but also provide a fully distributed model for the transactions to be executed (Figure 4.3).

The durability and reusability of the transaction and stability of the transaction (as the dynamic nature of SMEs) remains as some unanswered questions. It is important to avoid the abortion of the transaction even when some (or even all) participants are temporarily disconnected. This problem has been solved when one of participant (at each nested part of the transaction) is/are disconnected. But we try to provide a highly reliable environment which can cope with dynamicity of SMEs and high probability for their disconnections.

On the other hand, it is very important to keep the result of a successful or unsuccessful transaction (even by considering regular unavailability of initiator and other participants). Other issue is lookup

algorithm even more knowledge of unavailability of SMEs or their services based on their natural behaviour. This can be argued by the probability of fragmentation on such a network (D3.1). Our goal is not just providing a network structure for answering these requirements but also reusing fragmented network structures provided by the transactions to create a fully connected network.

4.1.3 Temporary Virtual Private Networks

As shown in Figure 4.3, the actual execution of a transaction creates a temporary network between service providers which will disappear after the transaction is finished. However, this network has some unique characteristics which are worth keeping for later use.

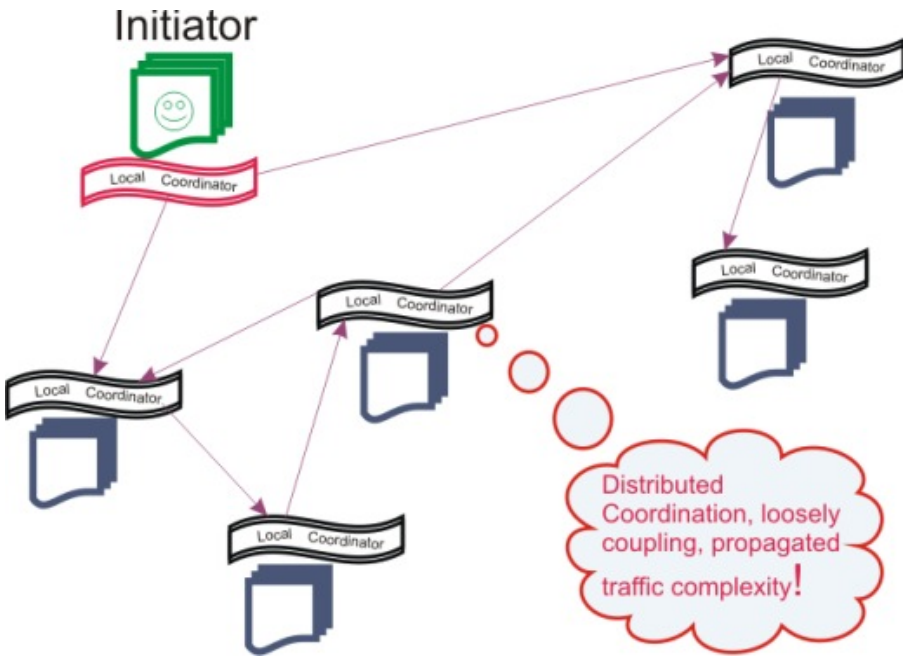


Figure 4.3. Local coordination on each platform

Apart of being fully distributed without any centralised control, it inherits all of the transaction model properties (D3.1 [RMK07]): in one hand, it is loosely-coupled which gives full local autonomy to platforms, and on the other hand it has resistance on failure and can be recovered - it can even handle short-term disconnections as the traffic is not focused on a centralised point. Meanwhile the platforms involved in a transaction are in a related domain and there is probability for them to do similar transactions again.

4.2 Virtual Private Transaction Networks

The temporary network created by a 'Transaction' is called *Virtual Private Transaction Network* (VPTN), as apart from the participants in the transaction, naturally they are shared with other platforms and normally they are not created as an actual network. Because of the specific properties of these networks we consider a component for keeping them and adding ability for re-using them.

Figure 4.4 shows a collection of these networks. Clearly, VPTNs are a collection of fragmented networks, apart from occasional overlaps between different transactions which is also transparent.

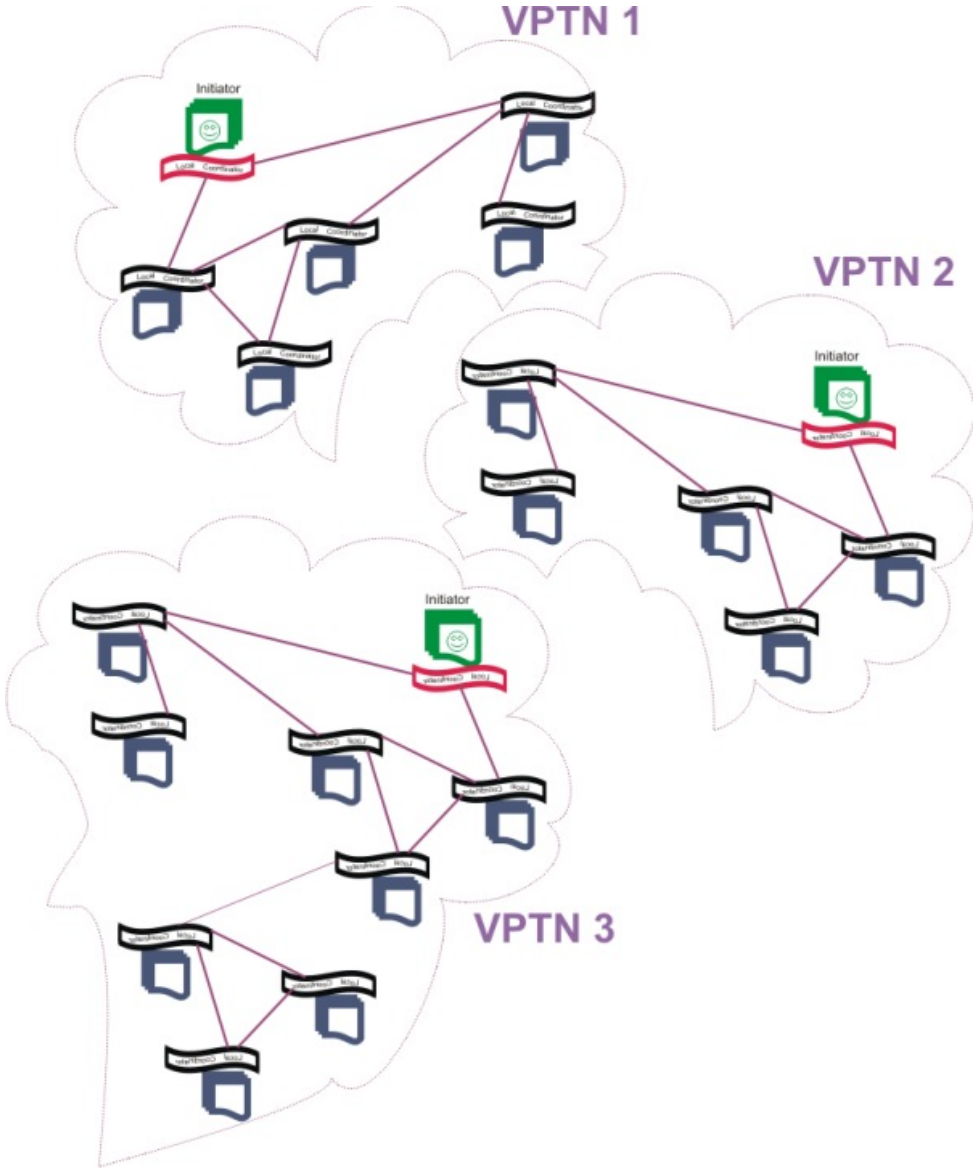


Figure 4.4. General overview of environment when several transactions are executed

4.2.1Global Service Repository

Actually the important duty of 'Global Service Repository' (introduce in section 3.1.4) is keeping these VPTNs. In the aspect of implementation, the simplest option which we considered is using a database but we consider the adoption of this to other forms (such as some standard schema which may show the connection and relationships). An example is given in Figure 4.5. The Local Coordinators are connected to the Global Service Repository, they can update links and information about other participants and their services can be saved. In this way, VPTNs can be saved permanently.

It is important to mention, keeping this information is not the only duty of 'Global Service Repository'. Actually 'Global Service Repository by using 'Web Service Information Investor', try to update its information about other platforms (web services) based on the architectural requirements (which may be clearer until end of this document) and at the same time, by using 'Web service Promoter', tries to transfer its records to the other platforms for creating more stable network (which will be explain in the rest of this chapter). Figure 4.5 can explain these connections between 'Global Service Repository' and 'Web service Promoters' and 'Web Service Information Investor'. There are several beneficial properties for this

architecture. We discuss some of them here and in the next deliverable we will analyse them in greater detail.

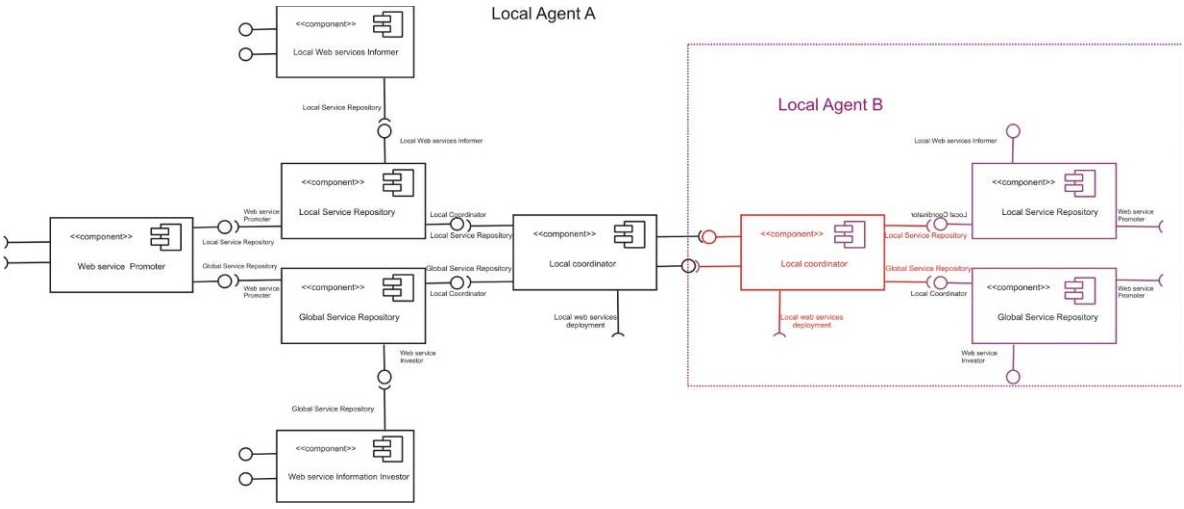


Figure 4.5. The role of 'Global Service Repository' in respect to Local Coordinator and VPTNs

4.2.2 Reliability and Result Prediction

One of the immediate benefits of considering the VPTNs is reliability and ability for prediction of the results on alternative scenarios. Based on the availability of platforms (services), a transaction can have more reliable execution and even during a temporary disconnection of a platform we can virtually keep the connection alive and give a timeout for return of the platform. The actual mechanism at the transactional level is implemented by the time-out lock (T-Lock) scheme introduced in deliverable D3.1 [RMK07].

Furthermore, the transaction effectiveness can be measured even before executing the transaction. If a service is not available (the service provider platform is disconnected, etc), the execution of alternative scenarios can be predicted and if the service is a vital service for the transaction (meaning that there is no alternative for that and success of the transaction depends on that specific service), the initiator can avoid executing the transaction altogether, since there is clear indication by the network itself that it will not be successful.

4.2.3 Losing valuable information

The current infrastructure avoids losing valuable information to some extent and provides practical methods for forward recovery (which has been discussed in view of the transaction model at a design level in deliverable D3.1 when we discussed the alternative coordinator during the failure or failing of achieving transaction/scenario condition, and from a more formal modelling perspective in Section 2.3.8 in this document). Even the result of an unsuccessful transaction can be saved and reused in restarting the transaction. On the other hand, keeping track of participants in the Global Service Repository gives us the ability to retrieve information about services and their platforms and we may need to repeat the same operation for that. Having information about web services of different participant and some information about availability of that (which supposed to be kept on 'Global Service Repository), can be helpful for starting the same transaction or even new transaction which share some parts of the unsuccessful transaction. Meanwhile it is possible to analyse the reasons of failure and tracking the operation of the transaction.

As each participant keeps the information about the others in its 'Global Service Repository', in this way the VPTN of a particular transaction is stable (and safe) during the transaction life time (even if some participants temporary disconnects from the network). An exceptional situation may occur when all participants (including the initiator) are disconnected at the same time. Unfortunately in these circumstances the reliability of the Global Service Repository can be questioned (as currently any update on a VPTN can be instigated by the participants of the transaction). For solving this problem, in the next sections we move on from a fragmented network to a de-fragmented network with some permanent clusters, which can guarantee the reliability of the stored information.

4.2.4Visualization

In our first attempt at visualisation, we have considered some simplifications which we hope by providing material from other partners of this work package (WP3) will gradually become more and more accurate:

- Geographical Distance: as we do not have specific data about geographical distance and its effects (such as time zone which it is quite effective factor on availability function and other parameters), we have had to use a random function. Figure 4.6 shows a simple example of such a distribution.
- Average or specific network traffic between SMEs, the average usage of web services between them.
- The number of potential candidates for stable nodes (or permanent online node, if there is such thing!).
- The mathematical model used for the formal analysis of the transaction behaviour described in Chapter 2 of this report is expected to be extended to fully cover all concepts of the distributed transactional model. Further improvements on aspects such as the minimum numbers of nodes in the network which can improve the final stability function, or even some alternative model from other partners which can fulfil the requirements of workpackage 3 (D3.1 contains a full specification of the requirements).
- Any specific improvement factor on the evolutionary framework. Our current evolutionary framework has added some values to the model which has been described in this report but work in this respect is in progress and the evolutionary model is open for additional modifications based on the network requirements.

Under these assumptions, we have started our visualisation by using a random geographical function and simplified time-zone for SMEs. In Figure 4.6, we demonstrate a sample of current design which it is based on our random assumptions.

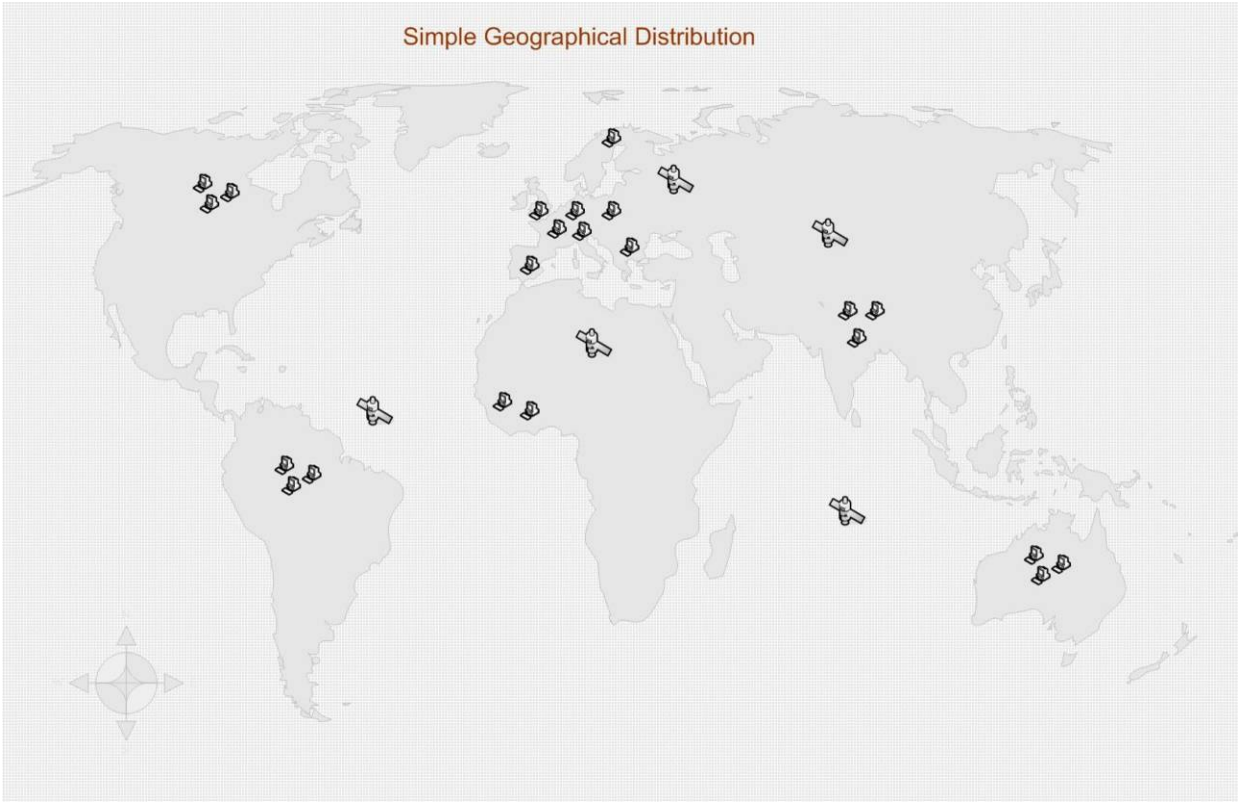


Figure 4.6. A sample of Geographical Distribution

4.3 Stable Private Networks

For reaching stability in each VPTN, we try to find some measurement of stability for each node in the network. Actually, it turns out that the probability for availability of each node is the important factor. For this reason, in this section we try to analyse this factor by using service availability and later on, we can use to ensure better stability in the network. Our purpose is to find a good candidate in each VPTN for keeping information about the private network in its repository and in this way have maximum stability of the network in time (for example, it may be able to keep this information for some hours!). Then in the next section we try to improve this network towards a fully permanent network.

4.3.1 Service Availability

As new characteristic, 'service availability' (which can be considered as part of *Quality of Service* QoS) is introduced. It is possible to get the regulation of platform/service availability which mostly is related to the regional time zone. For example specific SMEs in UK provide their services between 8:00AM and 17:00 GMT and this can be quite similar for other SME in UK. This regular availability can be derived by their business model or calculated by node neighbourhoods or simply provided by the SME itself.

In our model, service availability is like a promise from the business for being available on a specific time-frame regularly. This means we expect a service to be available in the time period which the business promised (as it is in this time frame it expects to do its transactions). Service availability can be dynamic

and changeable by the business. But the important point is that any unavailability during the service availability can reduce the stability measurement of the corresponding node.

4.3.2 Stability and Disconnections

As is shown in Figure 4.7, each SME has a property which shows the Expected Availability Time (EAT). Nevertheless, during this expected availability time it may disconnected. These disconnections will reduce stability (reliability) of the corresponding node in the final selection. This is the most important factor for considering the node as the most stable node in each VPTN. This stability can be simply calculated as below:

$$NodeStability = \frac{EAT - DisconnetionPeriods}{EAT}$$

As is clear $NodeStability \leq 1$ and a node whose NodeStability is closer to 1 is the most stable node in that specific VPTN and thus can keep the full and updated information about services and their providers in its repository.

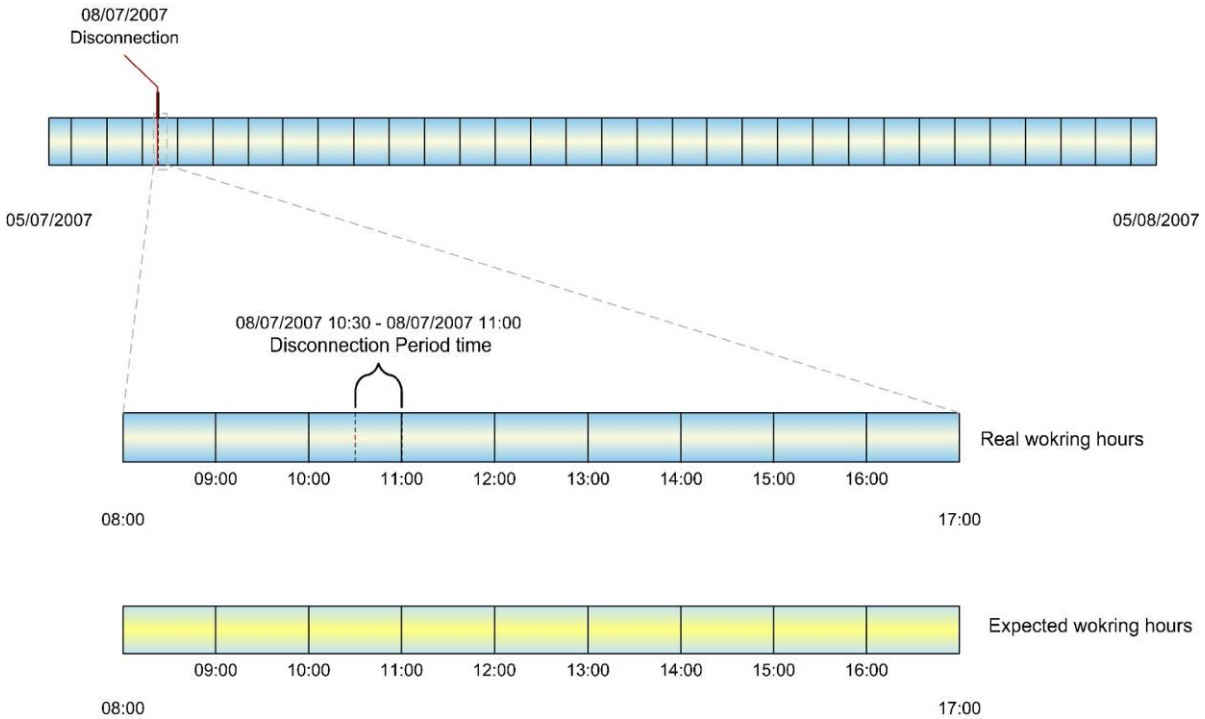


Figure 4.7 A node online timeline

4.4 Connected Network (Virtual Service Network)

In this section, we try to provide a fully connected network (without any fragmentation) by connecting VPTNs together. The best candidates for connecting VPTNs together are the most stable nodes in each VPTN. Figure 4.8 shows the demonstration of this connected network, in this deliverable, we mostly focus on the result of such a network (which shows the characteristics and necessities of it) and we assume

creating this network with the birth of OPAALS is feasible (in the next deliverable we will go through details of creation of the network by improving the birth model.

By connecting VPTNs, we mean 'Global Service Repository' of each candidate in each VPTN, will be connected to the candidate of the other VPTN. In this way we can improve the stability of the connected network (the maximum time for the network to be alive). However, we cannot warranty full stability of the network and still cannot avoiding the occasional fragmentation (because even in the best case, it is dependent on each platform's availability and if the total online time of all stable nodes cannot cover 24 hours, our network will collapse for some period of time, precisely that in which all of them are not available).

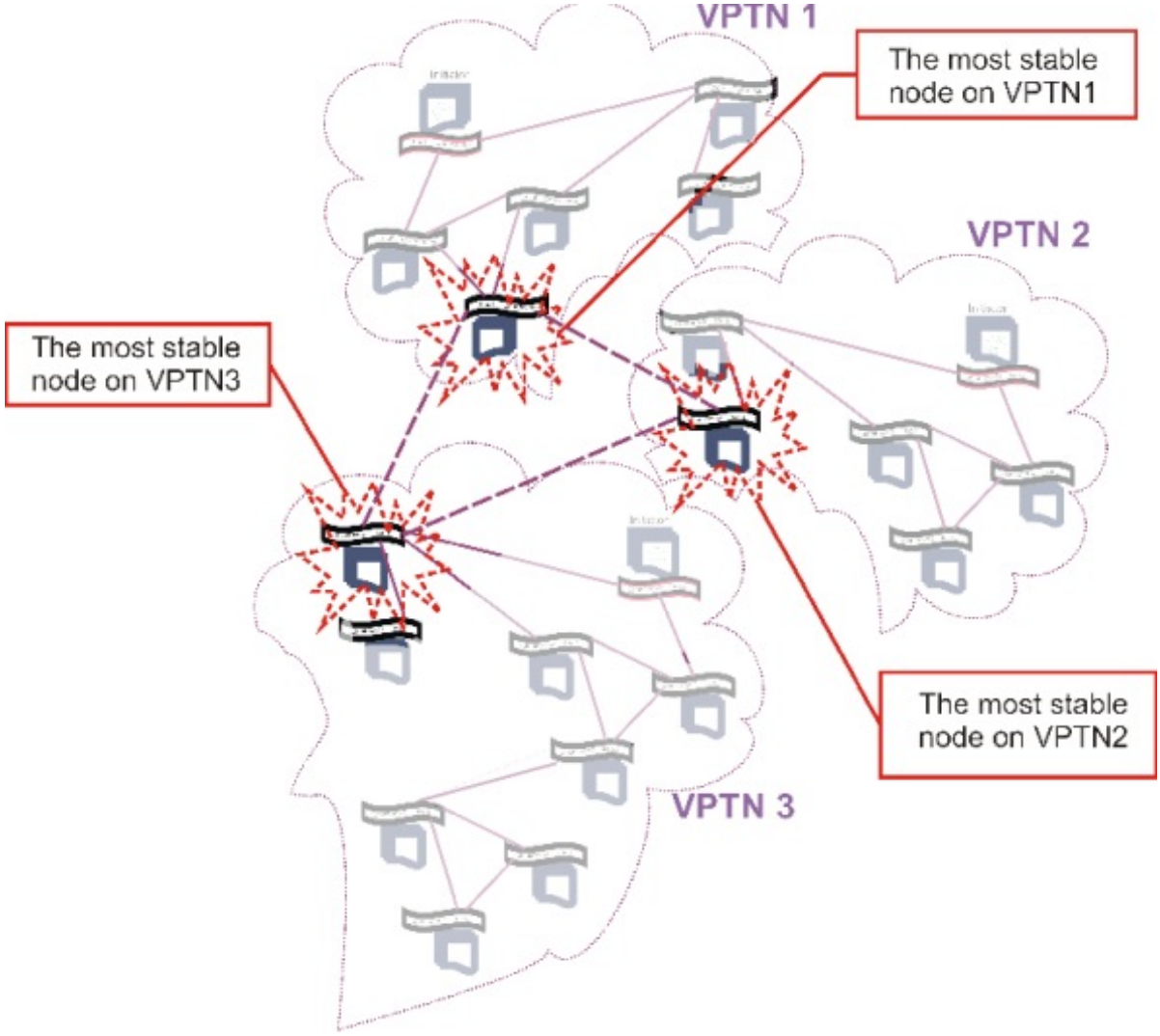


Figure 4.8: VSN (Connected VPTNs)

Figure 4.9, shows how nodes can be connected at the components level of our local agent based design discussed in Chapter 3. This shows the relationship between our model and the component based design of each local agent. The connection between nodes can be provided by two components: the Web service Promoter and the Web Service Information Investor.

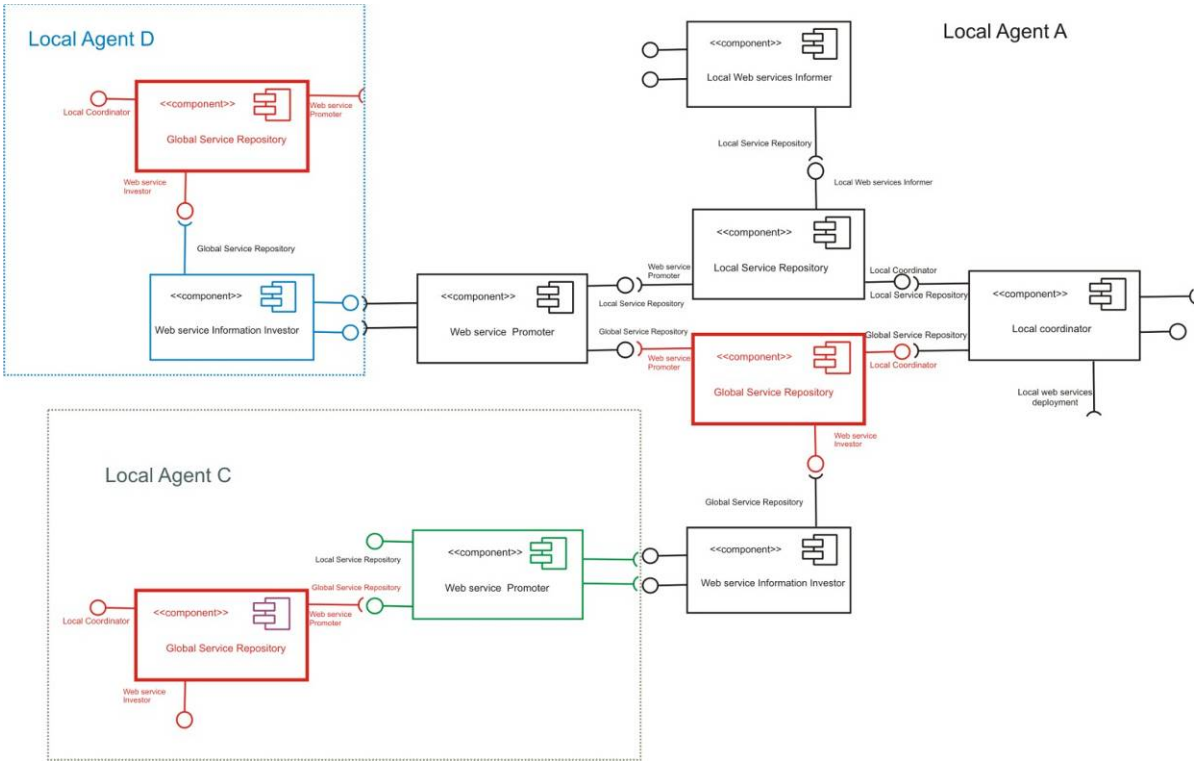


Figure 4.9. The network in practice

When any changes happen for some records of the local or the Global Service Repository, they use the Web service Promoter's interfaces and the Web service Promoter can use interfaces provided by the Web Service Information Investor in the other Local agent and Web Service Information Investor at that agent can update its Global Service Repository if needed (because in some cases it could be done already). As a result the Global Service Repository of an agent will use the same interfaces for Web service Promoter at the agent and this will be done for any agent connected to the Local agent. In this way, any changes on connected agents can be updated quickly. (More details can be found in the discussion in Section 3.1).

4.4.1 Permanent Node and Weaknesses

We have seen that by connecting the most stable nodes in each VPTN there is still considerable probability for fragmentation. It seems that the best solution is to use the most stable nodes which are permanently connected (their EAT is 24 hours). Figure 4.10 shows such nodes. In this figure, we have tried to shows different collection of SMEs, normal terminal-shapes presents small (and micro) enterprises/businesses which are not stable and their availability is very limited (their regulation for being part of the network will not follow their availability pattern) and server-shapes in the figure are small to medium enterprises which based on their business nature are available during a time period but their availability pattern is more stable than terminals and at the same time they are more available. And quadruple-server shapes are medium enterprises which bases on their business nature (for example a medium size 24-hours taxi agency with reasonable stability function) are available but that does not mean they are fully stable (they have more stable functionality).

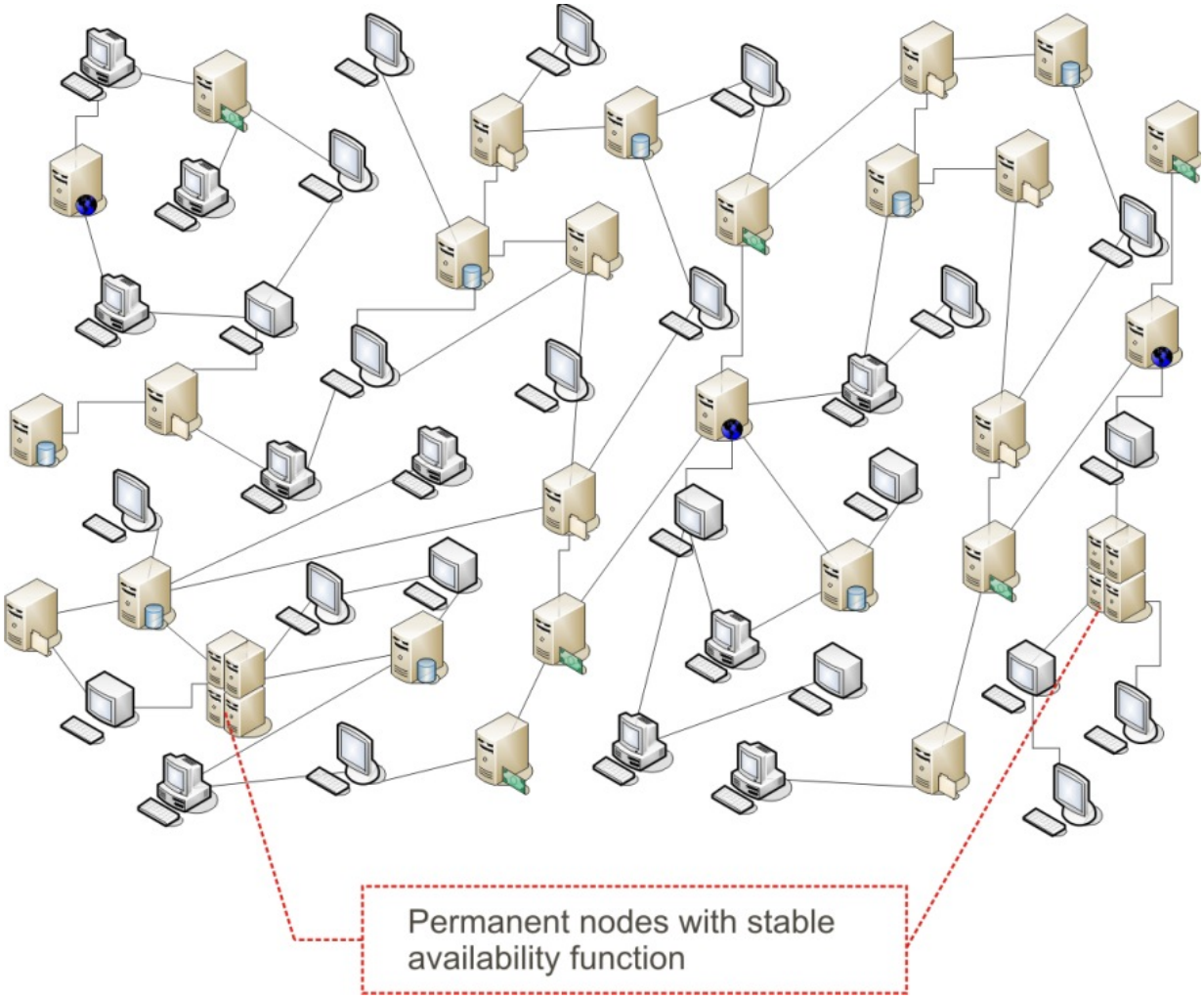


Figure 4.10: VSN with Permanent nodes

First of all, the number of these nodes in the network is quite low (if there are any). After finding such nodes, all most stable nodes from each VPTN must connect to such node(s) and perform regular updates through their Web Service Information Investor and Web service Promoter is necessary. Not only does this creates a costly traffic complexity but also we have to rely on few points of failure. That means the connected network is vulnerable to smart attacks (targeting the permanent nodes) and even accidental failures of permanent nodes are a significant risk.

Furthermore, peak-time and off-peak time for network traffic can make different problems. At the peak-time, traffic bottlenecks can reduce the performance dramatically while at the off-peak time the probability for fragmentation increases and the range of tolerance reduces. As time progresses, due to the strong dependency on permanent nodes, the topology will become completely static and finding new candidates for permanent nodes becomes more and more difficult. It is important to also consider the low performance of the expensive permanent nodes for their owners, since they come under increasing pressure (more traffic) in order to stabilise the network than doing any business of their own.

4.4.2 Virtual permanency and Permanent Clusters

As a first step towards moving to a more dynamic architecture which does not rely on just a few permanent nodes, we try to find permanent clusters on the network. For doing so, different time-zones for service availability should be detected and the most stable nodes in each VPTN of different time zones have to be selected.

The important part for finding permanent clusters is discovering different aggregations of these time zones which can cover 24 hours availability. Any union of the stable nodes of this aggregation (which provides 24 hour availability coverage) are actual permanent clusters. Figure 4.11, shows the simple situation in which two sets of time zones which can cover 24 hour service availability – the most stable nodes have been selected.

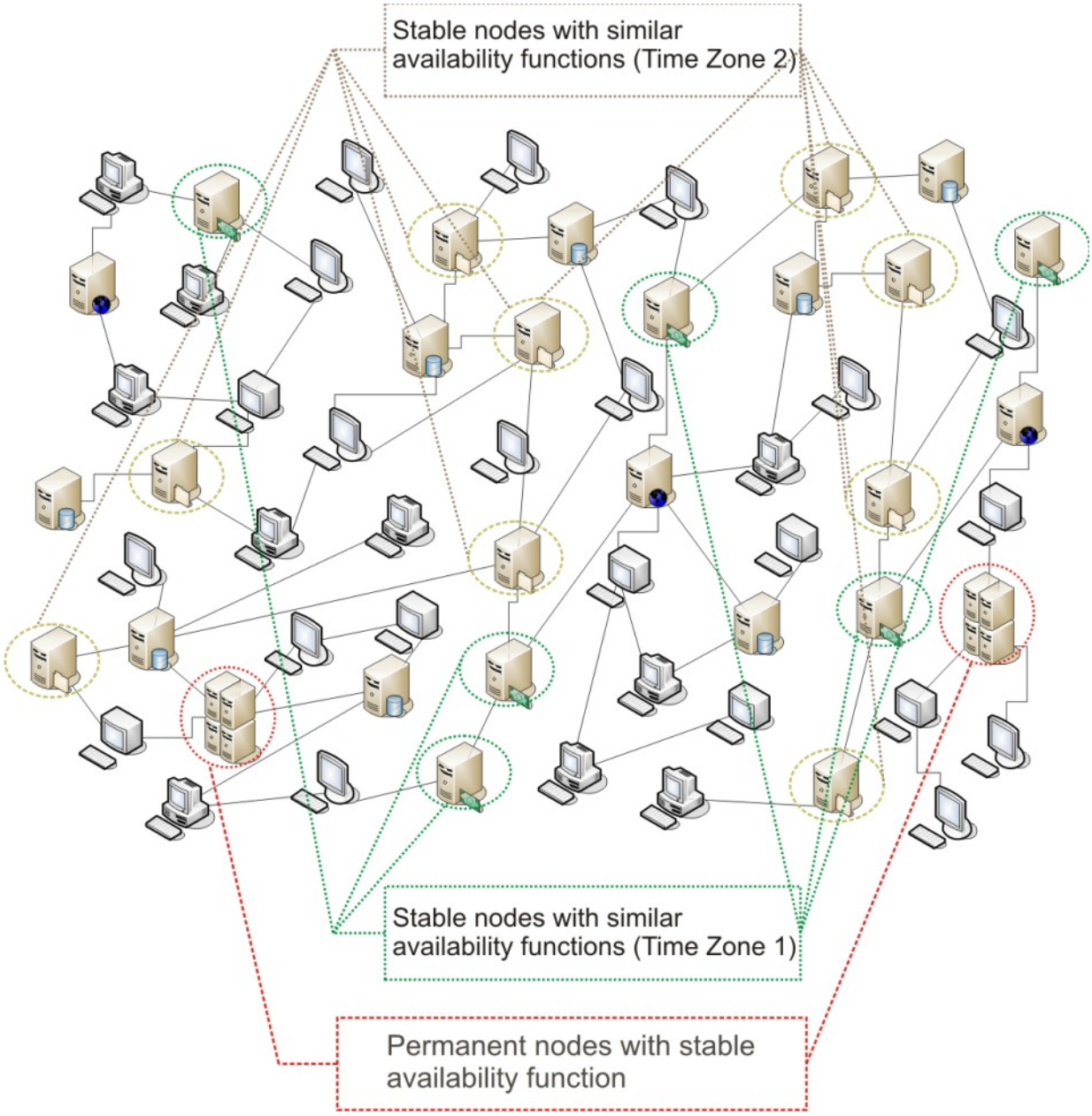


Figure 4.11: Stable nodes and virtual permanency

In Figure 4.11, there are different collections of SMEs. Similarly to Figure 4-10, terminal shapes presents small (and micro) enterprises/businesses which are not stable and their availability is very limited (their regulation for being part of the network will not follow their availability pattern) and server shapes are small to medium enterprises which based on their business nature are available during a time period but their availability pattern is more stable than terminals and at the same time they are more available, meanwhile servers with green sign have a good stability and are from the same time zone. On the other hand, servers with creamy sign similar to servers

with green sign, they have good stability but they are in different time zone (which these two time zones together can cover 24 hours of a day!). And quadruple-server shapes are medium enterprises which bases on their business nature (for example a medium size 24-hours taxi agency with reasonable stability function) are available but that does not mean they are fully stable (they have more stable functionality).

4.4.3 Virtual Super peers

As shown in Figure 4.12, by using stable nodes from permanent clusters, we can create Virtual Super Peers which can provide our desired stability for the network. Up to a level of reliability which we need, we can include redundant stable platforms from each available time zone. For example, in Figure 4.12 we have included two stable nodes from each time-zone (this increases the reliability).

In this manner, VPTNs can be connected and we can provide a connected network. Meanwhile the traffic is spread on VSPs, and since choosing stable nodes is a dynamic process (depending on EAT to Disconnection period of node during EAT) the topology can change from time to time. Furthermore, we expect a reasonable cluster coefficient (as each VPTN belongs to a transaction, they are in relevant domains and by connecting them to several VSPs, we actually increase the opportunity for that) and a fair distribution degree (as a result of propagating links to VSPs).

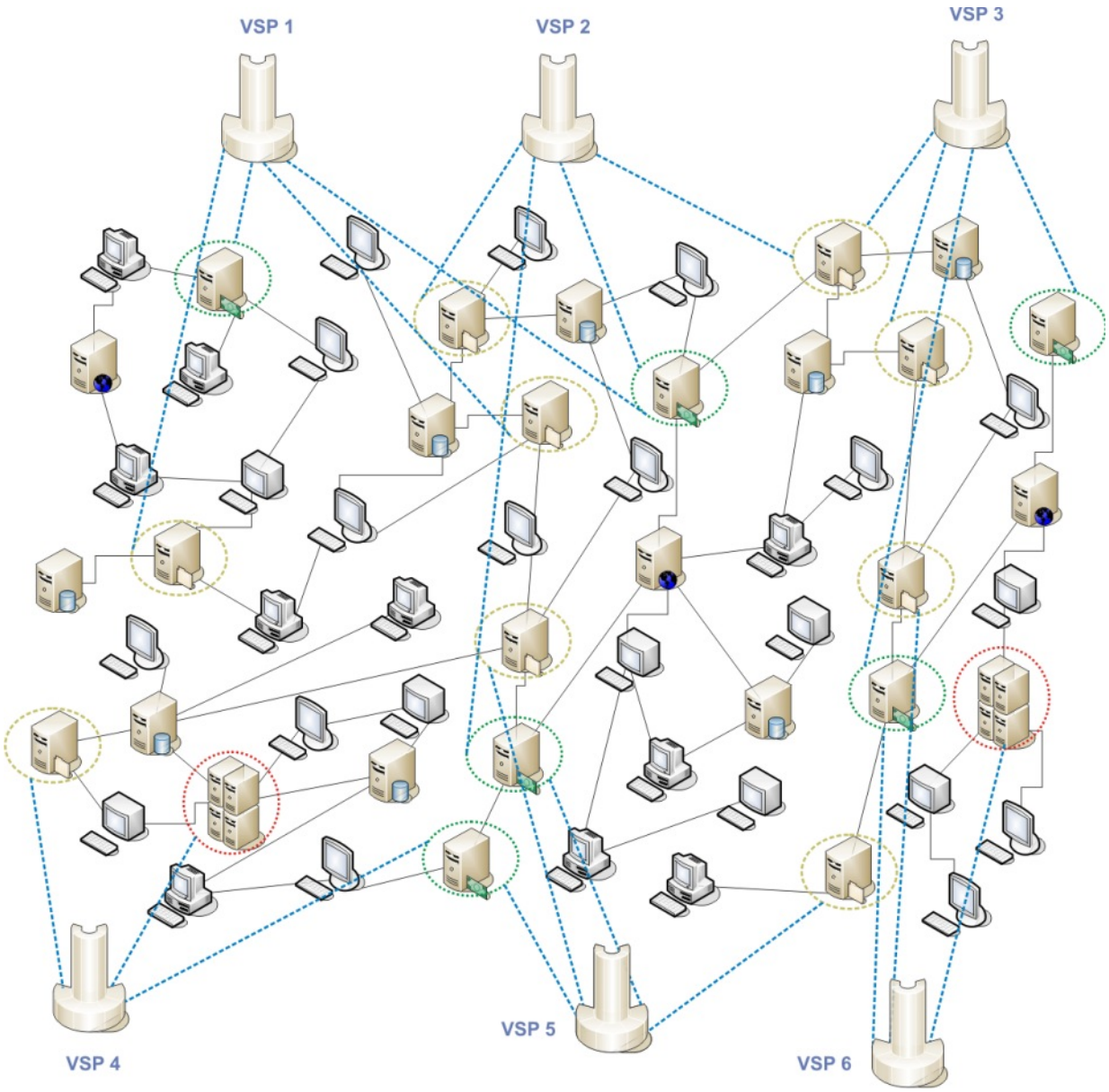


Figure 4.12: Virtual Super Peers

4.5 Evolutionary framework

The initial inspiration of the current model (described above) is drawn from the evolutionary framework. The primary requirement for an *autopoietic* P2P network is that there is no centralised directory as there is no precise control over the network topology or the placement of content/services on the network. It is formed by nodes joining the network following some loose rules.

The resulting topology has certain properties but, in contrast to structured designs, the placement of content is not based on any knowledge of this topology. To find content/services, a node queries its neighbours. The most common query method is *flooding* – the query is propagated to all neighbouring nodes within a certain radius. Unstructured designs are extremely resilient to nodes entering and leaving the system. We refer to the policy (model) which is applied when a node enters and/or leaves the network as the *birth and growth model*.

4.5.1 Requirements for evolutionary framework

It is important to clarify the priorities and then discuss reliability according to these priorities. In this way, we might be able to reach some level of formalisation or clear measurement of reliability within the network. In different scenarios/models, these priorities can be in conflict with each other and therefore, as a primary solution, we may need to engage in a trade-off (decrease the expectation in certain circumstances) between levels of reliability in view of different scenarios.

One of the major challenges in P2P network has to do with fragmentation – the situation where the network is divided to a number of smaller isolated networks ('islands'). Undoubtedly, the design of the *autopoietic* P2P network for OPAALS should take into account fragmentation as a top priority for reliability, but also for recoverability (most important in a business transactional environment, as discussed in D3.1 [RMK07] Ch. 2), and latency (especially with regard to content sharing), which are also considered to be high priorities.

4.5.2Connectivity

Analysing the distribution degree function alone can often mislead the reliability discussion. As a simple example, we may consider reliability in a typical scale-free network. In the first instance, the network failure tolerance and recoverability may indicate high resistance to fragmentation. However, a smart attack on hubs can have the directly opposite results, which may be missed in the first analysis of the network reliability. The situation is described in Figure 4.13.

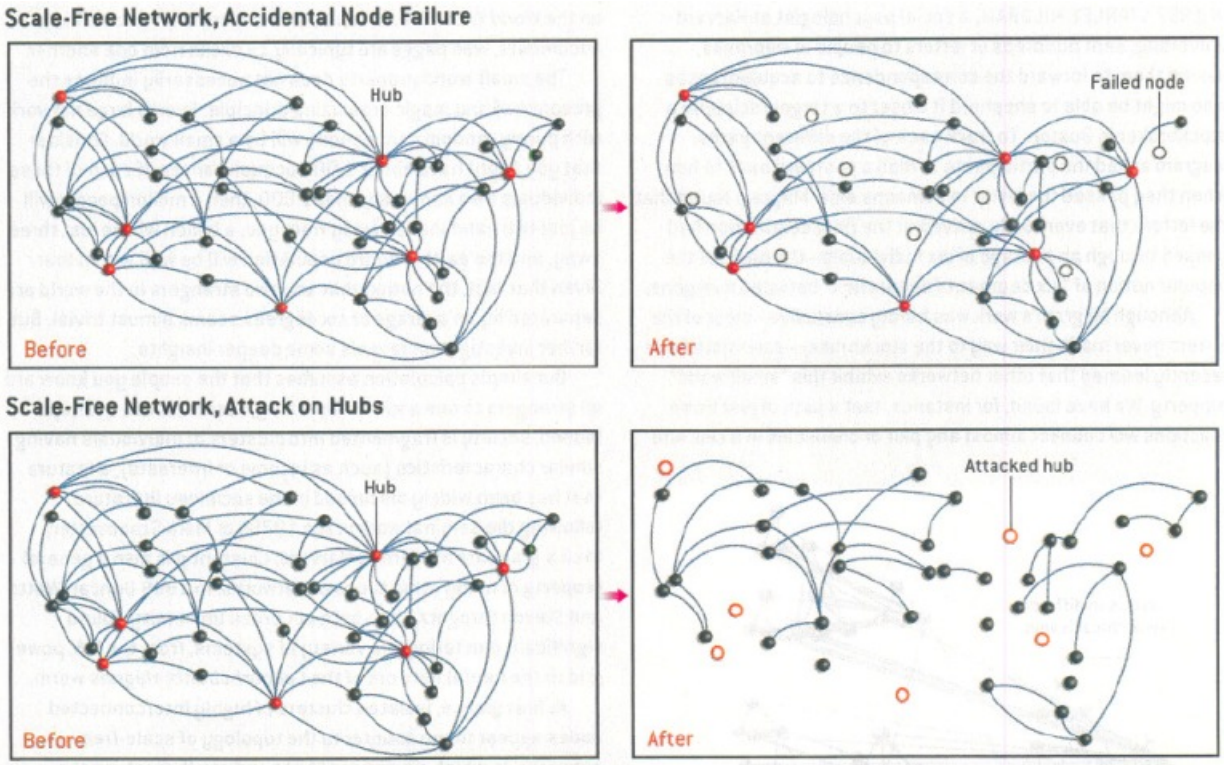


Figure 4.13 Fragmentation in scale-free networks

In this case, parameters such as network connectivity, edge propagation and minimum number of edges per node, can play an important role for resistance against fragmentation. Additionally, replication at

the level of network edges – that is, replication of edges instead of contents replication – can be considered as another important factor.

4.5.3General performance

Another puzzling concept in the *autopoietic* P2P network is latency. Traditionally, there has been a dispute regarding the role of concurrency in relation to latency (see Figure 4.14). But this argument is mainly valid when we are dealing with any level of centralised control/processing in which any preceding transaction (request) has to be followed and checked by some centralised authority. This may be a simple identity check or the application of a complicated centralised concurrency control mechanism for ensuring consistency.

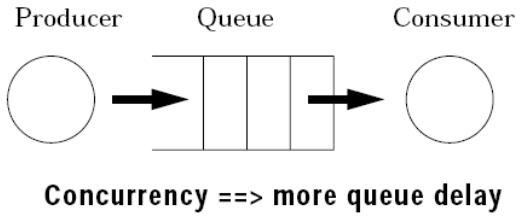


Figure 4.14 Latency

In contrast with this view, the *autopoietic* P2P network neither relies on a centralised authority, nor suffers from global topological control. Furthermore, the *autopoietic* P2P network is supposed to be fully distributed and this implies that each node can act as a distributed unit for processing requests and improving latency. Consequently, allowing for more transactions to execute concurrently not only does not deteriorate latency but also seems necessary for recuperating the performance (latency) and exploiting the potential processing power of the system (recall D3.1 Chapter 2, especially the discussion in Section 2.4.2).

If the *autopoietic* P2P network for OPAALS is to be considered for content/knowledge sharing too, and not only as a P2P network which provides a high performance transactional environment for SMEs, then another factor may play a crucial role too: the handling of data packages.

The loss or delay of data packages will have a direct impact on the response time, as shown in Figure 4.15, which means the middle-ware nodes for routing (or even data processing) have a significant role to play with regard to latency. Of course, any traffic bottlenecks and unreliable nodes at the heart of the network have the foremost impact on the latency.

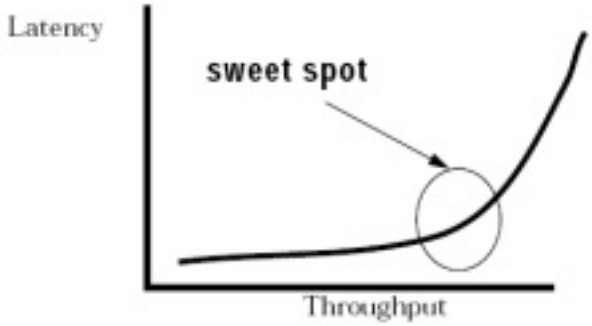


Figure 4.15. Latency in relation to throughput

4.5.4Evolutionary candidate model

Considering the above requirements, we propose to draw upon the evolutionary growth of metabolic (signal transduction) networks, as studied in the work of Rzhetsky and Gomez [RzG01], [GLR01], in designing the birth and growth model for the *autopoietic* P2P network in OPAALS. The analysis by Rzhetsky and Gomez shows that the evolutionary growth of metabolic networks has scale-free characteristics while it also has some interesting properties with respect to network connectivity.

The frequency of vertices connected to exactly k other vertices in metabolic (signal transduction) networks follows a power-law distribution. On the other hand, the distribution function degree is equivalent to

$$P(k) \sim k^{-\gamma}$$

or, precisely

$$P(k) = c.k^{-\gamma}$$

when c is a normalizing constant and γ diverges across networks (but usually has a value between 1 and 3) [BaA99].

The network follows a fractal model, as the shape of this distribution remains invariant to changes in network scale. This means that a small sub-graph has the same distribution as the complete graph from which it is derived. As the total number of different DNA and protein domains in a genome with both the total number of genes and the overall network topology, make an equation [KWR+02], it is statistically possible to analyse the relationship between the edge duplication/creation and the network topology.

4.5.5Growth of metabolic networks

Based on the Rzhetsky and Gomez hypothesis [RzG01], [KWR+02], [GNR03] in molecular networks each edge is implemented as a pair of mutually specific molecular structures. For example, in the case of an edge corresponding to phosphorylation of protein B by protein A, the beginning of an edge in protein A is encoded as a kinase domain, whereas the ending of the same edge in protein B is encoded as a specific protein domain recognized by the kinase domain of protein A.

In another example, the beginning of an edge is encoded as a DNA-binding domain of a transcription factor, whereas the ending of the same edge at a gene regulated by the transcription factor is encoded by a DNA motif specifically recognized by the DNA-binding domain.

As a result, Rzhetsky and Gomez, have introduced a directed graph in which up the direction of the edge (towards the node or opposite) is given in ‘upstream domain’ and ‘downstream domain’ terms. Each vertex of a molecular network is assumed to have either both upstream and downstream domains, or just one of the domains.

Furthermore, the authors advocate that upstream and downstream domains experience independent duplication and that after duplication, each new domain copy randomly picks a domain from the available pool of domains of the opposite type (upstream or downstream) to form a two-domain protein. If there is no domain of the opposite type available, a one-domain protein is formed.

In the Rzhetsky and Gomez model, a domain is defined as a functional unit that provides a specific interaction between two molecules. Note that, in the case of downstream domains, their definition lumps together protein and DNA domains so as to include network edges that correspond to transcription activation/inhibition events. In this analysis they use the InterPro and Pfam databases to identify protein domains.

4.5.6Domain duplication or link replication

In this model, the number of upstream and downstream domains in each class has been monitored (without analysing how these domains are combined into genes or proteins). As a general approach for modelling stochastic processes, the primary assumption is that evolution in the network is governed by a homogeneous continuous-time Markov process, in which individual changes in the network arrive spontaneously but at constant rates. We note that this assumption may need to be changed/refined to model more realistic chains of events in the *autopoietic* P2P network, but nevertheless provides a valuable starting point.

One of the major evolutionary events that affect network growth is the duplication of genes and proteins (Figure 4.16). We can model domain duplication as some level of replication, the so-called link replication, which mostly inherits the nodes' links of the original copy. As a result of applying link replication, the connectivity of the network will be increased.

If we want to make some similarity with our model; 'upstream' can show availability of a node in particular in the Global Service Repository while 'downstream' shows the node is addressed by another node (who pointed to that node and has the address of node on its 'Global Service Repository'). Actually this model has been the main inspiration for the structure and behaviour of the Global Service Repository.

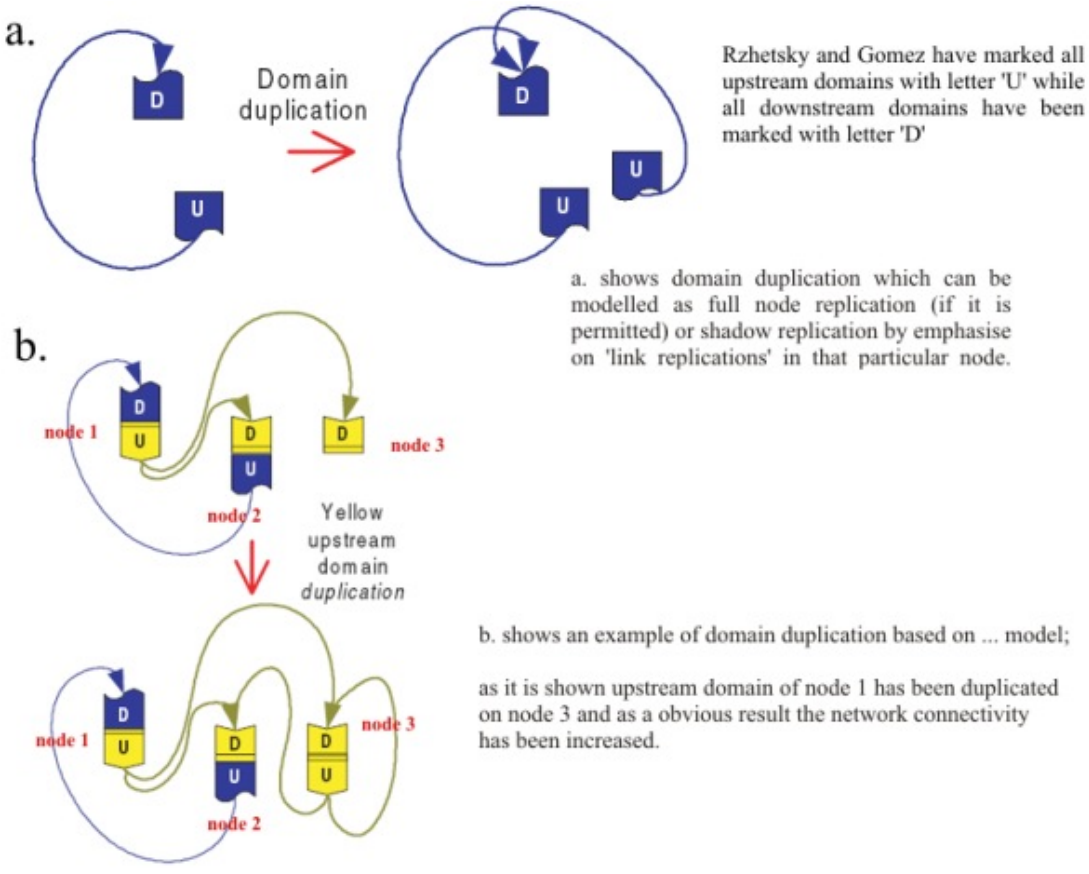


Figure 4.16. Domain duplication with some modifications from [GNR03]

The example shown in Figure 4.16 part b (creating a new edge from node 3 to node 2) easily can be interpreted in our model as adding the address (or some web services) of node 2 in node 3's Global Service Repository.

4.5.7 Edge innovation: adding a node to the network

The second type of major event in evolution is the birth of new classes of edges (see Figure 4.17). This is often called *edge innovation*. Similarly to the evolutionary growth of metabolic networks [KWR+02], [RzG01], the birth of a new class of edges can happen during the addition of a new node to the network. Innovation can also be applied during the process of de-fragmentation – that is, when a fragmented network would try to connect to the other fragmented part of the network.

In this paradigm, adding a new node can be started by innovation of new class of edge to that particular node (Figure 4.17 can be helpful in this respect) and in our local agent structure, this can be simple insert a record to 'Global Service Repository' of main node (who is pointed to the new node).

After that domain duplication can provide more reliable connectivity between the new node and the rest of the network (at least reverse connection, which having an upstream from new node to the rest of the network). In term of de-fragmentation, innovation of new class of edges between two nodes of fragmented networks can be consider as the solution (in evolutionary framework point of view) and that can be tighter by domain duplication in the next step.

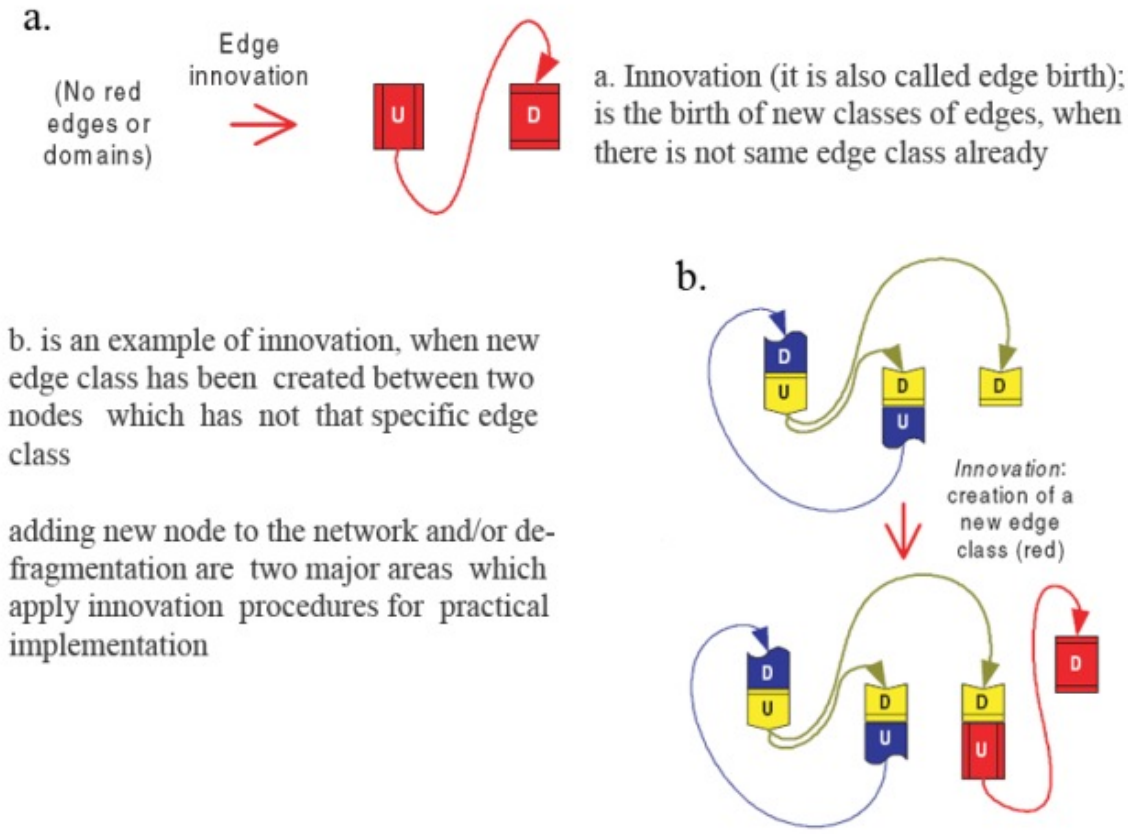


Figure 4.17. Edge innovation with some modifications from [GNR03]

The adaptation of basic concepts from the model for the evolutionary growth of metabolic networks, together with the two basic events in evolution – namely, domain duplication and edge innovation – form the foundation for a birth and growth model for the *autopoietic* P2P network in OPAALS. Following this biologically-inspired approach (apart from being within the cross-disciplinary spirit of the OPAALS community), allows for interesting characteristics of this model, such as good connectivity and stability of the resulting scale-free network (with high resistance against hubs attack because of distribution degree and fractal nature of the network), can be inherited in the P2P network. As an advantage this model provides, the total number of network vertices is at least three times as large as the number of genes, (based on the analysis by Rzhetsky and Gomes)

$$D(t) \geq 3 \bullet G \frac{\sum_{i=1}^{V} i^{-\gamma_{in}}}{\sum_{j=1}^{V} j^{-\gamma_{in}+1}}$$

where the coefficient γ may vary approximately from 2 to 3 for most metabolic networks, G is the number of genes and D is the number of pairs of distinct domains. This means that network stability can be increased in terms of connectivity, as with increasing the network size (number of nodes), the connectivity between nodes increases faster (numbers and classes of edges). This is one of the reasons for applying the model to our main P2P network.

4.5.8 Unconstructive innovation: removing a node from the network

In the original evolutionary growth of metabolic networks, domain duplications and edge innovations are applied based on a probability function (which can be calculated statistically), and there is no proper model for losing class edges. In contrast with that, dynamicity of SMEs' behaviours shows they may disconnect from and join the network regularly – this may even happen as a pre-defined behaviour, such as opening business hours. This is one reason we have started to make some extension to the model governing the evolutionary growth of metabolic networks.

In the first instance we may reimburse unconstructive innovation by applying domain duplication. In this way, not only does the connectivity function does not act in an adverse way but also the scale-free properties may be enhanced (higher distribution degree on the stable nodes which do not disconnect regularly).

Figure 4.18 shows an example of unconstructive innovation in which a node after losing an edge will be supported by its neighbours (by supporting here we mean that other nodes apply domain duplication on that particular node resulting in more upstream and downstream edges to and from the node which lost a neighbour).

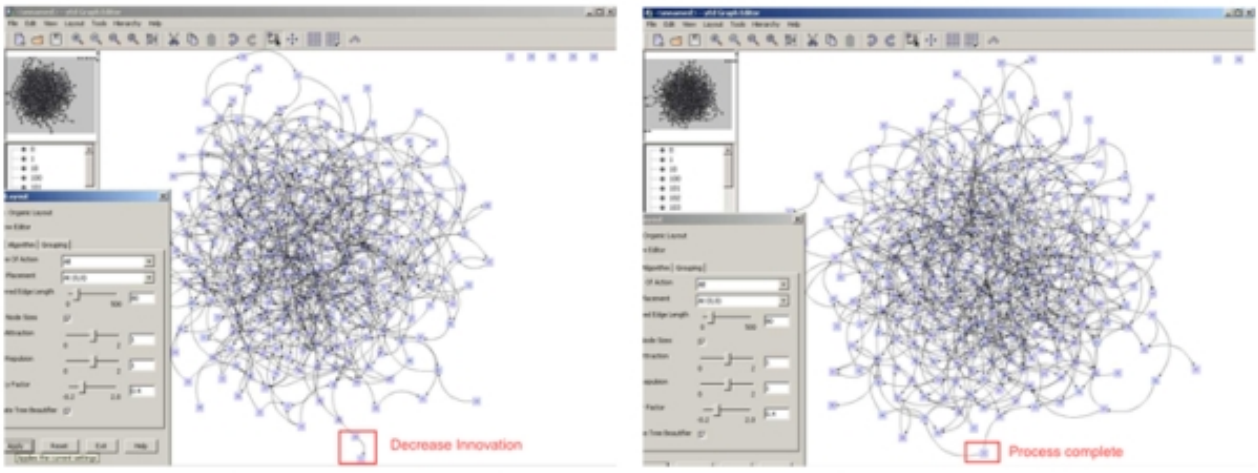


Figure 4.18. Unconstructive Innovation

Therefore, when a node is on the system for long, even if it loses neighbours, its connectivity will increase. As an inspiration of this model, we have added this to our first design of the *autopoietic* P2P

network (see D3.1 [RMK07] for more details). As a result, the probability function used by the original evolutionary growth of metabolic networks for domain duplications and edge innovations cannot be applied directly. In any case, we are looking forward to seeing the statistical study of other partners to have better estimation of SMEs pattern behaviour in the real-world and as a result re-calculate the probability function for all three major events in the network (domain duplication, edge innovation and unconstructive innovation). For avoiding unpredictable instability, we have introduced the availability function and remodelled domain duplications based on that (recall Sections 4.3 and 4.4).

5 Concluding Remarks

In this report we have described work related to the key target (objective 6) of OPAALS which is concerned with the development of a P2P architecture to support distributed long-running transactions in open communities of SMEs. A large number of transaction are expected to take place in a digital ecosystem for business and a large number of SMEs are expected to participate (joining and leaving the network periodically) if the proposed solution can foster an environment in which open and trusted collaborations can be conducted that ensure their sustainability within a pan-European digital ecosystem. This implies that a large number of (critical) interactions will take place over the P2P network, and thus there is a need for a formal analysis on both fronts – the distributed transaction model and the underlying P2P network characteristics.

Whilst deliverable D3.1 [RMK07] focused on the design of a distributed transaction model for digital ecosystems and a purely distributed peer-to-peer network to support it, in this deliverable we have been concerned with the formal analysis of the behaviour of long-running transactions and the underlying service compositions involved, as well as the performance analysis of the P2P network that supports distributed long-running transactions, including aspects of the evolutionary nature such architecture should have.

We gave a brief account of the nature of the (mostly, but not exclusively, business) activities expected to take place within a digital ecosystem and outlined a number of surrounding issues that frame the concept of a transaction in this context. Long-running business transactions have only recently attracted the interest of the formal methods community within computer science, and we reviewed different models that have been proposed for capturing the behaviour of a long-running transaction. This revealed that a number of (important) aspects have not been adequately considered and we described a formal model of long-running transactions that goes some way to addressing some of these issues.

In particular, we advocated a vector language based description of a transaction that can be used to formally describe the behaviour patterns the underlying service compositions should follow when the transaction is executed. The proposed model, in its current state, can capture dependencies within a transaction (release of data between its subtransactions, orderings according to heuristics or business processes themselves). In addition, the formal description of long-running transactions we gave in Chapter 2, is firmly based on well established computer science theories. An immediate benefit of this is the ability to describe parallel actions faithfully, in the sense of true-concurrency rather than simply composition of independent sequential actions. Another side effect is that the resulting model is rather generic, meaning that there is no foreclosure for describing dependencies across transactions (e.g. partial results) and work is under way in this direction.

Note that this would place the transaction model in a very good position within the asynchronous-messaging Ajax world, which is the centrepiece of Web 2.0, but also critical for adopting this technology for the OPAALS OKS. The issue of asynchronous communication within a webpage and the asynchronous calls to the server, for example, can be informed by the formal model presented in Chapter 2 (Section 2.3) which is expressive enough to capture multiple access points and the dependencies that arise when communication occurs between them.

As stated in our first deliverable (D3.1) of Workpackage 3, we are considering SOA as the enabling technology for a digital ecosystem for business. This means that our design of both the transaction model and the supporting P2P network should respect the primary characteristics of SOA. The challenge of coordinating long-running multi-service transactions in a fully distributed manner led to the development of a local agent whose structure was described in detail in Chapter 3. The proposed solution paved the way for starting the implementation of the model and this was demonstrated by a small scenario involving the booking of a taxi. More complicated scenarios and possible ramifications are being considered at the time of writing and the preliminary results are encouraging and will be reported in the following deliverables.

The analysis of a simple scenario for implementation also facilitated the overall analysis and design of the network, with regard to the transaction support it is meant to provide. It became apparent that keeping information about previous transactions and, perhaps even more importantly, of the temporary smaller P2P network over which they took place, can be used to inform the evolution of the topology of the overall P2P architecture. Results of this analysis including the arrival at a stable network which can avoid fragmentation and the usage of a temporary network initially created by individual transactions were discussed in Chapter 4, among with issues of service and node availability as well as dealing with temporary disconnections. Further experimentation with the evolution model inspired by the networks of cell metabolism is needed to arrive at the best candidate parameters for increasing the performance.

We also described the evolution model nominated for the evolutionary framework behind the proposed P2P architecture. We showed what were the necessary changes and improvements for its adaptation and outlined how we have gone about applying this model to our network. Another interesting aspect which deserves further attention is that of reusing the results of previous transactions as the starting point for realising *explorative* composition, since this seems to be the starting point for composing services on-the-fly. Work is in progress in both respects and shall be reported on in our next deliverables.

6 References

[ABC+01] Aitken, D., Bligh, J., Callanan, O., Corcoran, D., & Tobin, J., (2001), Peer-to-Peer Technologies and Protocols, Available at: http://ntrg.cs.tcd.ie/undergrad/4ba2.02/p2p/ (last accessed: 02/11/06).

[AbH02] Aberer, K. and Hauswirth, M. An Overview on Peer-to-Peer Information Systems, Workshop on *Distributed Data and Structures (WDAS-2002)*, (available at: minoas.di.uoa.gr), 2002.

[Ada99] Adamic, L. A. The Small World Web. Proceedings of *Third European Conference on Research and Advanced Technology for Digital Libraries (ECDL 99)*, Paris, France. Springer Verlag, Lecture Notes in Computer Science vol 1696, pp. 443-452, 1999.

[AdH00a] Adamic, L. A. and Huberman, B. A. Power-law Distribution of the World Wide Web. *Science*, 287 (5461): p. 2115, 2000.

[AdH00b] Adamic, L. A. and Huberman, B. A. The Nature of Markets in the World Wide Web. *Quarterly Journal of Electronic Commerce*, 1(1): 5-12, 2000.

[AJB00] Albert, R., Jeong, H. and Barabási, A. Error and Attack Tolerance in Complex Networks, *Nature*, 406: 378-382, 2000.

[ALH01] Adamic, L. A., Lukose, R. M., Puniyani, A. R. and Huberman, B. A. Search in Power-law Networks, Physical Review E, Vol 64, p. 046135 , 2001.

[ALH02] Adamic, L. A., Lukose, R. M. and Huberman, B. A. Local Search in Unstructured Networks, arXiv:cond-mat/0204181 v2, 4 Jun 2002.

[AzM03] Azzedin, F. and Maheswaran, M. Trust Modeling for Peer-to-Peer Based Computing Systems, Proceedings of 17th *International Parallel and Distributed Processing Symposium* (IPDPS'03), pp: 99.1, 2003.

[Bag05] Baghdadi, Y. A Web Services-based Business Interactions Manager to Support Electronic Commerce Applications", Proceedings of the *7th International Conference on Electronic Commerce*, Vol. 113, pp: 435-445 , 2005.

[BHT04] Balakrishnan, H. Hwang, I. and Tomlin. C. (2004), 'Polynomial approximation algorithms for belief matrix maintenance in identity management.' In Proceedings of the 43rd IEEE Conference on Decision and Control, Atlantis, Paradise Island, Bahamas, December 2004.

[BAJ99] Barabasi, A. L., Albert, R. and Jeong, H. The Diameter of the World Wide Web, arXiv:cond-mat/9907038 v1, 2 Jul 1999.

[BaA99] Barabasi A. L and Albert R. Emergence of Scaling in Random Networks. *Science* 286(5439): 509–512, 1999.

[Bar03] Barabasi L. A. *Linked: How Everything is Connected to Everything Else and What it Means for Business, Science and Everyday Life*. PLUME, USA, 2003.

[Bar01] Barkai, D. Peer-to-Peer Computing, Technologies for Sharing and Collaborating on the Net. Intel Corporation, 2001.

[BeN97] Bernstein P. A. and Newcomer E. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, 1997.

[Ber03] Bergner, M. Improving Performance of Modern Peer-to-Peer Services, , Department of Computer Science, UMEA University, 2003.

[BHMcC+03] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C. and Orchard, D. Web Services Architecture, 8 August 2003. Available at: http://www.w3.org/TR/2003/WD-ws-arch-20030808/

[BoS06] Bowles, K-F., J; Moschoyiannis, S.; Concurrent Logic and Automata Combined: a Semantics for Components. In *Proc. CONCUR 2006 – Formal Foundations of Coordination Languages and Software Architectures (FOCLASA'06),* Electronic Notes in Theoretical Computer Science, 175(2): 135-151, Elsevier B. V., 2007.

[Bro02] Broberg, J. C. Glossary for the OASIS WebService Interactive Applications (WSIA/WSRP), OASIS Web Services Interactive Applications TC Committees, 2002. Available at: http://www.oasis-open.org/committees/wsia/glossary/wsia-draft-glossary-03.htm

[BMM05] Bruni, R.; Melgratti, H.; Montanari, U.; Theoretical Foundations for Compensations in Flow Composition Languages. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pp. 209-220, ACM Press, 2005.

[Buc03] Buchanan M. *Small World: uncovering nature's hidden networks*. Weidenfeld and Nicolson, London, 2002.

[BEK93] Bukhres, O. A., Elmagarmid, A. K. and Kühn, E. Implementation of the Flex Transaction Model. *IEEE Data Engineering Bulletin*, 16(2): 28-32, 1993.

[BPE] Business Process Engineering Language (BPEL) Specification, version 1.1. Availabel at: http://www.ibm./com/developerworks/library/ws-bpel

[BCF+02] Butler, M.; Chessell, M.; Ferreira, C.; Griffin, C., Henderson, P. and Vines, D.; Extending the Concept of Transaction Compensation. *IBM Systems Journal*, 41(4):743-758, 2002.

[BuF04] Butler, M. and Ferreira, C.; An Operational Semantics for StAC, a Language for Modelling Long-running Business Transactions. In *Proc. Coordination 2004*, Lecture Notes in Computer Science, Vol. 2949, pp. 87-104, Springer, 2004.

[BHF05] Butler M. J, Hoare A. C. R. and Fereira C. A Trace Semantics for Long-Running Transactions. *Proceedings 25 Years of CSP*, Lecture Notes in Computer Science, Vol. 3525, pp. 133-150, Springer, 2005.

[CCC+03] Cabrera, L. F., Copeland, G., Cox, W., Feingold, M., Freund, T., Johnson, J., Kaler, C., Klein, J., Langworthy, D., Nadalin, A., Orchard, D., Robinson, I., Shewchuk, J., Storey, T. and Satish Thatte. Web Services Coordination Framework (WS-Coordination), September 2003.

[CCC+04a] Cabrera L. F. Copeland G., Cox W., Feingold M., Freund T., Johnson J., Kaler C., Klein J. Langworthy D., Nadalin A., Orchard, D., Robinson I., Shewchuk J., Storey T. and S. Thatte. (2004-a) Web Services Atomic Transaction Framework (WSAtomicTransaction), January 2004.

[CCC+04b] Cabrera L. F., Copeland G., Cox W., Feingold M., Freund T., Johnson J., Kaler C., Klein J., Langworthy D., Nadalin A., Orchard D.; Robinson, I., Shewchuk J., Storey T. and S. Thatte. (2004-b) "Web Services Business Activity Framework (WSBusinessActivity)", January 2004

[CCJ+04] Cabrera, F. L., Copeland, G., Johnson, J. and Langworthy, D. Coordinating Web Services Activities with WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity. Available at: http://msdn.microsoft.com/webservices/default.aspx, January 2004.

[CdeS95] Campos, R. V., de Souza e Silva, E. Availability and Performance Evaluation of Database Systems Under Periodic Checkpoints, Proceedings 25th *International Symposium on Fault-Tolerant Computing (IEEE CNF)*, 27-30, pp: 269 – 277, 1995.

[CDF+03] Ceponkus, A.; Dalal, S.; Fletcher, T.; Furniss, P.; Green, A. and Pope B. Business Transaction Protocol Version 1.0, An OASIS Committee Specification, 3 June 2002. Available at: http://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf

[ChG94] Chen, Y.-W.; Gruenwald, L. Research Issues for a Real-time Nested Transaction Model. Proceedings of the *IEEE Workshop on Real-Time Applications*, pp:130 – 135, 21-22 July 1994.

[CKM+03] Curbera F.; Khalaf R.; Mukhi, N.; Tai, S. and Weerawarana, S. The Next Step in Web Services, *Communications of the ACM*, 46(10): 29-34, October 2003.

[CMH+02] Clarke, I., Miller, S.G., Hong, T.W., Sandberg, O. and Wiley, B. Protecting Free Expression Online with FreeNet", *IEEE Internet Computing*, 6(1) 40-49, Jan-Feb 2002.

[CML04] Clark, T. Martin, S. and Liefeld, T. (2004) 'Globally distributed object identification for biological knowledgebases', BRIEFINGS IN BIOINFORMATICS. VOL 5. NO 1. pp: 59–70. MARCH 2004.

[CrP02] Crowcroft, J. and Pratt I. Peer-to-Peer: Peering into the Future. In Proceedings of the *IFIP-TC6 Networks 2002 Conference*, Pisa, Italy, May 2002.

[CSW+00] Clarke, I., Sandberg, O., Wiley B. and Hong, T. W. FreeNet: A Distributed Anonymous Information Storage and Retrieval System, In Proceedings of the *ICSI Workshop on Design Issues in Anonymity and Unobservability,* Berkeley, California, June 2000.

[CTT97] Chin, C. Chung-Kie Tung and Shang-Rong Tsai. Phaeton: a log-based architecture for high performance file server design. Proceedings of *Pacific Rim International Symposium on Fault-Tolerant Systems*, pp. 28-33, 1997.

[DaP90] Davey, B. A. and Priestley, H. A. *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[Dat96] Date C. J. *An Introduction to Database Systems*. 5th Edition, Addison Wesley, USA, 1996.

[DBE06] Digital Business Ecosystems (DBE) EU-FP6 IST Integrated Project No 507953. Available at http://www.digital-ecosystem.org [last accessed 19 March 2007]

[DKA+04] Derbas, G., Kayssi, A., Artail, H. and Chehab, A. TRUMMAR - a trust model for mobile agent systems based on reputation. Proceedings *IEEE/ACS International Conference on Pervasive Services* (ICPS 2004), pp. 113 – 120, 2004.

[DKR02] Druschel, P.; Kaashoek, M. F. and Rowstron, A. I. T. Peer-to-Peer Systems. Proceedings of *First International Workshop, IPTPS*, 2002.

[DLD97] Domel, P.; Lingnau, A.; Drobnik O. Mobile Agents Interaction in Heterogeneous Environment, *First International Workshop on Mobile Agents*, April 1997, No 1219, pp: 136-148

[DNB03] Ding, C. H.; Nutanong, S.; Buyya, R. Peer-to-peer Networks for Content Sharing, Technical Report, GRIDSTR-2003-7, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, December 2003.

[DSW94] Deacon, A.; Schek, H.-J.; Weikum, G. Semantics-based multilevel transaction management in federated systems. Proceedings of *10th International Conference on Data Engineering (IEEE CNF)*,, pp. 452 – 461, 1994.

[DTL+03] Dalal, S.; Temel, S.; Little, M.; Potts, M. and Webber, J. Coordinating Business Transactions on the Web, *IEEE Internet Computing* 7(1): 30-39, January - February 2003.

[EDB+05] ElMoustapha Ould-Ahmed-Vall, Douglas M. Blough, Bonnie S. Heck, George F. Riley (2005) 'Distributed Global Identification for Sensor Networks', School of Electrical and Computer Engineering, Georgia Institute of Technology Atlanta, GA 30332-0250.

[EdL95] Eder, J. and Liebhart, W. The Workflow Activity Model WAMO. Proceedings of *3rd Int. Conference on Cooperative Information Systems (CoopIS)*, 1995.

[Edw02] Edwards, J. *Peer-to-Peer Programming on Groove*, Addison-Wesley, 2002.

[Elm94] Elmagarmid A. *Database Transaction Model for Advanced Applications*, Morgan – Kaufmann, 1994.

[EmT00] Emmerich, W.; Tai, S. Advanced Transactions in Enterprise Java Beans, Proceedings of *Engineering Distributed Objects: Second International Workshop*, pp: 215, Springer Berlin / Heidelberg, 2000.

[FDF+04] Furnis, P.; Dalal, S.;Fletcher, T.; Green, A.; Ceponkus, A. and Pope, B. Business Transaction Protocol version 1.1.0, Committee Draft, November 2004. Available at: http://www.oasis-open.org/committees/download.php/9836/business_transaction-btp-1.1-spec-wd-04.pdf

[FML+97] Fraga, J. Maziero, C. Lung, L.C. Loques Filho, O.G. Implementing Replicated Services in Open Systems Using a Reflective Approach, Proceedings of *Third International Symposium on Autonomous Decentralized Systems* (*ISADS'97*), 9-11 April 1997, pp: 273 – 280, 1997.

[FrI01] Fritzke, U., Jr. Ingels, P. (2001), "Transactions on partially replicated data based on reliable and atomic multicasts", 21st International Conference on Distributed Computing Systems, April 2001, page(s): 284 – 291

[FuG05] Furnis, P. and Green, A. (2005), "Choreology Ltd. Contribution to the OASIS WS-TX Technical Committee relating to WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity", OASIS WS-TX Technical Committee, 16 November 2005. (avalaible at www.oasis-open.org/committees/download.php/15808/Choreology.WS-TX.TC.Contribution.2005-11-16.doc , last availability; 06.June.2006).

[GAP04] Gilbert, A. Abraham, A. Paprzycki, M. (2004), "A system for ensuring data integrity in grid environments", ITCC 2004. International Conference on Information Technology: Coding and Computing, 5-7 April 2004, pp: 435 - 439 Vol.1

[GaW01] Gallagher D. Wilkerson. R. "Network performance statistics for university of south carolina." In http://eddie.csd.sc.edu, 2001

[GiC00] Gillies J., Cailliau R. (2000), How the Web Was Born: The Story of the World Wide Web, Oxford University Press, England

[GLR01] Gomez,S.M., Lo,S.H. and Rzhetsky,A. (2001) "Probabilistic prediction of unknown metabolic and signal-transduction networks." Genetics, Vol. 159, 1291-1298, November 2001

[G-MG+91]Garcia-Molina, H.; Gawlick, D.; Klein, J.; Kleissner, K.; Salem, K. (1991), "Coordinating activities through extended sagas: a summary", Compcon Spring '91. Digest of Papers (IEEE CNF), 25 Feb.-1 March 1991, pp: 568 – 573

[G-MS87] Garcia-Molina, H. and Salem, K., (1987), "Sagas", Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987, pp: 249-259

[GNR03] Gomez S., Noble W. S. and Rzhetsky A., "Learning to predict protein–protein interactions from protein sequences", BIOINFORMATICS, Vol. 19 no. 15 2003, pages 1875–1881

[Gon01] Gong, L. (2001), "JXTA: a network programming environment", Internet Computing, IEEE, Volume: 5, Issue: 3, May/Jun 2001, pp: 88-95.

[Gud04] Gudgin, M. (2004), "Secure, reliable, transacted: innovation in Web Services architecture", Proceedings of the 2004 ACM SIGMOD international conference on Management of data (ACM Press), 2004, pp: 879 – 880

[Hag95] Haghjoo M. S. (1995), Transaction Actors in Cooperative Information Systems, PHD Thesis, The Australian National University, Australia

[Hag96] Haghjoo M. S. (1996), "Scheduling and Scripting Mega Transaction", in prcds of the 2nd International Computer Conference, Iran Computer Society, Amir Kabir University, Iran

[Hag97] Haghjoo M. S. (1997), "Compensating Mega Transaction", in prcds of the 3rd International Computer Conference, Iran Computer Society, Iran University of Science and Technology, Iran

[HaP92] Haghjoo, M.S.; Papazoglou, M.P. (1992), "TrActorS: a transactional actor system for distributed query processing", Proceedings of the 12th International Conference on Distributed Computing Systems (IEEE CNF), 9-12 June 1992, pp: 682 – 689

[HBT04] Hwang, I. Roy, K. Balakrishnan, H. and Tomlin C. (2004) 'A Distributed Multiple-Target Identity Management Algorithm in Sensor Networks', 43nd IEEE Conference on Decision and Control, Atlantis, Paradise Island, Bahamas, December 2004.

[Hoa85] Hoare C. A. R. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HPS93] Haghjoo, M.S.; Papazoglou, M.P.; Schmidt, H.W. (1993), "A semantic-based nested transaction model for intelligent and cooperative information systems", , Proceedings of International Conference on Intelligent and Cooperative Information Systems (IEEE CNF), 12-14 May 1993, Pages:321 – 331

[H-RC02] Hong-Ren Chen; Chin, Y.H. (2002)," An efficient real-time scheduler for nested transaction models", Ninth International Conference on Parallel and Distributed Systems (IEEE CNF), 17-20 Dec. 2002, pp: 335 – 340

[HST01] Herrero, J.L.; Sanchez, F.; Toro, M.(2001), "Fault tolerance as an aspect using JReplica",FTDCS 2001. Proceedings. The Eighth IEEE Workshop on Future Trends of Distributed Computing Systems, 31 Oct.-2 Nov. 2001, pp: 201-207

[HYP01] Heuvel, W-J Van; Yang, J. and Papazoglou, M.P. (2001). Service Representation, Discovery, and Composition for E-Marketplaces, Proc. Of International Conference on Cooperative Information Systems cooPIS01),Sep, 2001.

[JSM01] Janaki Ram, D.; Chandra Sekhar, N.S.K.; Uma Mahesh, M.(2001), "A data-centric concurrency control mechanism for three tier systems", 2001 IEEE International Conference on Systems, Man. and Cybernetics, Volume: 4 , 7-10 Oct. 2001, pp:2402 - 2407 vol.4

[JTK89] Jenq, B.-C.; Twichell, B.C.; Keller, T.W.(1989), "Locking performance in a shared nothing parallel database machine", IEEE Transactions on Knowledge and Data Engineering, Volume: 1 , Issue: 4 , Dec 1989 pp: 530 – 543

[Kan01] Kan, G., (2001), Gnutella, Peer-to-Peer: Harnessing the Power of Disruptive Technologies, A. Oram (ed.), O'Reilly Press, USA.

[KaX92] Kakeshita, T.; Haiyan Xu (1992), "Transaction sequencing problems for maximal parallelism", Second International Workshop on Transaction and Query Processing (IEEE), 2-3 Feb. 1992, pp: 215 – 216

[KuR03] Kurose, J. F. & Ross, K. W. (2003), Computer Networking: A Top-Down Approach Featuring the Internet, Addison Wesley, Boston, USA.

[KWR+02] Karev G., Wolf Y., Rzhetsky A., Berezovskaya, F. and Koonin E., "Birth and death of protein domains: A simple model of evolution explains power law behavior", BMC Evolutionary Biology 2002, 2:18, 14 October 2002

[LaP01] Lala, C.; Panda, B. (2001), "Evaluating damage from cyber attacks: a model and analysis", IEEE Transactions on Systems, Man and Cybernetics, Volume: 31 , Issue: 4 , July 2001, pp: 300 – 310

[LCC+02] Lv, Q.; Cao, P.; Cohen, E.; Li, K. and Shenker S. (2002), "Search and replication in unstructured peer-to-peer networks", Proceedings of the 16th international conference on Supercomputing, 2002, pp: Pages: 84 – 95

[Lie06] Lieberman, B. (2006), "SOA transaction management -- Part I: The transaction coordination service;", developer Works on SOA and Web services at IBM Ltd,15 May 2006. Available at: http://www-128.ibm.com/developerworks/rational/library/may06/lieberman/

[LiF03] Little, M. and Freund, T. (2003), "A comparison of Web services transaction protocols", developer Works on SOA and Web services at IBM Ltd., 07 October 2003. Available at: http://www-128.ibm.com/developerworks/webservices/library/ws-comproto/

[LiT96] Liang, D.; Tripathi, S.K. (1996), "Performance analysis of long-lived transaction processing systems with rollbacks and aborts", IEEE Transactions on Knowledge and Data Engineering, Volume: 8, Issue: 5, Oct. 1996, pp: 802 – 815

[LiZ04] Limthanmaphon, B. and Zhang, Y. (2004), "Web service composition transaction management", Proceedings of the fifteenth conference on Australasian database, vol. 27, 2004, pp: 171-179

[LLW+03] Guo Qiong Liao, Yun Sheng Liu, Li Na Wang. Chu Ji Peng (2003), "Concurrency control of real-time transactions with disconnections in mobile computing environment", International Conference on Computer Networks and Mobile Computing (IEEE CNF), 20-23 Oct. 2003, pp: 205 – 212

[LoC04] Losee, R.M.; Church, L. (2004), "Information retrieval with distributed databases: analytic models of performance", IEEE Transactions on Parallel and Distributed Systems, Volume: 15 , Issue: 1, Jan. 2004 pp:18 – 27

[Mad97] Madria, S.K. (1997), "Concurrency control algorithm for an open and safe nested transaction model", Proceedings of 1997 International Conference on Communications and Signal Processing (IEEE CNF), 9-12 Sept. 1997, pp: 907 - 912 vol.2

[Maz77] Mazurkiewicz, A. Concurrent Program Schemes and their Interpretation. Technical report DAIMI PB-78, Aarhus University, 1977.

[Maz88] Mazurkiewicz, A. (1988) Basic Notions of Trace Theory. In de Baker, de Roever and Rozenberg, eds, *Proceedings of Linear time, Branching Time and Partial Orders in Logics and Models for Concurrency*, Lecture Notes in Computer Science, Vol. 354, pp. 285-363, Springer-Verlag, 1988

[M-EGB98] Munoz-Escoi, F.D. Galdamez, P. Bernabeu-Auban, J.M. (1998), "ROI: an invocation mechanism for replicated objects", Seventeenth IEEE Symposium on Reliable Distributed Systems, 20-23 Oct. 1998, pp: 29 – 35

[Mil01] Miller, J. (2001), "Jabber: Conversational Technologies." Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Andy Oram, editor. O'Reilly and Associates, Sebastopol, California. 2001.

[Mil80] Milner A.J.R. Calculus for Communicating Systems. Lecture Notes in computer Science, Vol. 92, Springer Verlag, 1980.

[MoS04] Moschoyiannis, S. Shields, M. W. (2004) A Set-Theoretic Framework for Component Composition. *Fundamenta Informaticae* 59(4): 373-396, 2004.

[MSK05] Moschoyiannis, S., Shields, M. W. and Krause, P. J. Modelling Component Behaviour using Concurrent Automata. In *Proceedings ETAPS 2005 – Formal Foundations of Embedded Software and Component-Based Architectures (FESCA'05)*, Electronic Notes in Theoretical Computer Science, Vol. 141, pp. 199-220. Elsevier, 2005.

[Mos05] Moschoyiannis, S. Specification and Analysis of Component-Based Software in a True-Concurrent Setting. PhD Thesis, University of Surrey, 2005.

[MKS07] Moschoyiannis, S., Krause, P. J. and Shields, M. W. A True-Concurrent Interpretation of Behavioural Scenarios. In *Proceedings ETAPS 2007 – Formal Foundations of Embedded Software and Component-Based Architectures (FESCA'07)*, Electronic Notes on Theoretical Computer Science, Elsevier, 2007. *To appear*

[Mos85] Moss J. E. B. *Nested Transactions: an Approach to Reliable Distributed Computing*. MIT Press, USA, 1985.

[MNS+04] Mustafa, M.D. Nathrah, B. Suzuri, M.H. Abu Osman, M.T. (2004), "Improving data availability using hybrid replication technique in peer-to-peer environments", AINA 2004. 18th International Conference on Advanced Information Networking and Applications, 2004, pp: 593 - 598 Vol.1

[MRW+93] Muth, P.; Rakow, T.C.; Weikum, G.; Brossler, P.; Hasse, C. (1993), "Semantic concurrency control in object-oriented database systems", Ninth International Conference on Data Engineering (IEEE CNF), 19-23 April 1993, pp: 233 – 242

[NeC02] Nektarios, G. Christodoulakis, S. (2002), "UTML: Unified Transaction Modeling Language", Proceedings of the Third International Conference on Web Information Systems Engineering (IEEE CNF), 12-14 Dec. 2002, pp: 115 – 126

[Nod93] Nodine, M.H. (1993), "Supporting long-running tasks on an evolving multidatabase using interactions and events", Proceedings of the Second International Conference on Parallel and Distributed Information Systems (IEEE CNF), 20-22 Jan. 1993, pp: 125 – 132

[NoZ94] Nodine, M.H.; Zdonik, S.B. (1994), "Automating compensation in a multidatabase", Proceedings of the Twenty-Seventh Hawaii International Conference on Software Technology (IEEE CNF), Volume: 2, 4-7 Jan. 1994, pp: 293 – 302

[NPW81] Nielsen, M.; Plotkin, G.; Winskel, G.; Petri Nets, Event Structures and Domains, part 1. *Theoretical computer Science*, 13:85-108, 1981.

[PaG03] Papazoglou M.P. and Georgakopoulos, D. (2003), "SERVICE-ORIENTED COMPUTING", COMMUNICATIONS OF THE ACM, October 2003/Vol. 46, No. 10, pp:24-28

[PaH05] Papazoglou, M.P.; van den Heuvel, W.-J. (2005), "Web services management: a survey", IEEE Internet Computing, Volume: 9, Issue: 6, Nov.-Dec. 2005, pp: 58- 64

[PaH99] Pavlova,E.; van Hung, D. (1999), "A formal specification of the concurrency control in real-time databases", Software Engineering Conference, 1999. (APSEC '99) Proceedings. Sixth Asia Pacific, 7-10 Dec. 1999, pp: 94 – 101

[Pap03] Papazoglou M. P. (2003) , "Service-Oriented Computing: Concepts, Characteristics and Directions", 4th International Conference on Web Information Systems Engineering (WISE'03) , Rome, Italy, 2003.

[PDB+97] Papazoglou, M.; Dells, A.; Bouguettaya, A.; Haghjoo, M. (1997) "Class library support for workflow environments and applications", IEEE Transactions on Computers, Volume: 46 , Issue: 6, June 1997, pp: 673 – 686

[PDH+96] Papazoglou, M.P.; Delis, A.; Haghjoo, M.; Bouguettaya, A. Language support for long-lived concurrent activities. Proceedings of the *16th International Conference on Distributed Computing Systems* (IEEE CNF), 27-30 May 1996, pp: 698 – 705, 1996.

[Plo00] Plonka. D. "Uw-madison napster traffic measurement." http://net.doit.wisc.edu/data/Napster/, March 9 2000

[PoH01] Posselt, D.; Hillebrand, G.(2001), "Database support for evolving data in product design", The Sixth International Conference on Computer Supported Cooperative Work in Design (IEEE CNF), 12-14 July 2001, pp: 377 – 383

[PTD+06] Papazoglou M. P., Traverso P., Dustdar S., Leymann F. and Kramer B. J. Service-Oriented Computing Research Roadmap. In *Dagstuhl Seminar Proceedings 05462, Service-Oriented Computing (SOC)*, pp. 1-29, 2006. Available at: http://drops.dagstuhl.de/opus/volltexte/2006/524 [Last accessed: 19 Apr 2006]

[QXL02] Qiujing Li; Hong Xu; TingLong Liu (2002), "Coordinating transaction model in CDBMS", The 7th International Conference on Computer Supported Cooperative Work in Design, 25-27 Sept. 2002, pp: 421 – 425

[RaN99] Ramampiaro, H.; Nygard, M.(1999), "Cooperative database system: a constructive review of cooperative transaction models", International Symposium on Database Applications in Non-Traditional Environments (DANTE '99), IEEE CNF, pp: 315 – 324

[Raz99] Razavi A. R. (1999), "Design and Implementation of Recovery Management for Mega Transaction and Model implementation", MSC Thesis, Iran University of Science and Technology, Iran

[RFH+01] Ratnasamy, S.; Francis, P.; Handley, M.; Karp, R. and Shenker, S. (2001), "A scalable content-addressable network", Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, 2001, pp: 161 – 172

[RiB06] Ripon, S.; Butler, M.; Relating Semantic Models of Compensating CSP. Technical Report, School of Electronics and Computer Science, University of Southampton, 2006.

[Rip01] Ripeanu M. (2001), "Peer-to-Peer Architecture Case Study: Gnutella Network", First International Conference on Peer-to-Peer Computing (P2P'01), 2001.

[RKM06] A. R. Razavi, P. J .Krause and S. K. Moschoyiannis. DBE Report D24.5: DBE Distributed Transaction Model, University of Surrey, 2006.

[RLF02] Ripeanu, M.; Iamnitchi A. and Foster I. (2002) "Mapping the Gnutella Network" Internet Computing, IEEE, Volume: 6, Issue: 1, Jan/Feb 2002, pp: 50-57.

[RKM06] Razavi R.; Krause, P.J.; and Moschoyiannis S.; Digital Business Ecosystem Transaction Model. DBE Project Report D24.28, University of Surrey, 2006.

[RMK07] Razavi R.; Moschoyiannis S.; and Krause, P.J. Preliminary Architecture for Autopoietic P2P Network focusing on Hierarchical Super-Peers, Birth and Growth Models. OPAALS project Deliverable D3.1, 2007.

[RMK07a] A. R. Razavi, S. K. Moschoyiannis and P. J Krause. A Coordination Model for Distributed Transactions in Digital Business Ecosystems. In Proc. IEEE Int'l Conf on Digital Ecosystems and Technologies (IEEE-DEST'07). IEEE Computer Society, 2007.

[RMK07b] A. Razavi, P. Malone, S.Moschoyiannis, B.Jennings, P.Krause. A Distributed Transaction and Accounting Model for Digital Ecosystem Composed Services. In Proc. IEEE Int'l Conf on Digital Ecosystems and Technologies (IEEE-DEST'07). IEEE Computer Society, 2007.

[RoD01] Rowstron A. and Druschel P. (2001), "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", Proceedings of the eighteenth ACM symposium on Operating systems principles, 2001, pp: 188 – 201

[RzG01] Rzhetsky A. and Gomez1 S., "Birth of scale-free molecular networks and the number of distinct DNA and protein domains per genome", BIOINFORMATICS, Vol. 17 no. 10 2001, Pages 988–996

[ScP03] Schmidt, C.; Parashar, M. (2003), "Flexible information discovery in decentralized distributed systems", 12th IEEE International Symposium on High Performance Distributed Computing, 2003. Proceedings. 22-24 June 2003, pp:226- 235

[SCW+04] Siau, K.L.; Chan, H.C.; Wei, K.K. (2004), "Effects of Query Complexity and Learning on Novice User Query Performance With Conceptual and Logical Database Interfaces", *IEEE Transactions on Systems, Man and Cybernetics,* Volume: 34 , Issue: 2 , March 2004, pp: 276 – 281

[SEN06] Software Engineering for Service-Oriented Overlay Computers (SENSORIA). EU-FP6 IST Integrated Project. Available at: http://sensoria.fast.de/ [Last accessed: 19 march 2007].

[SHD+01] Segun, K.; Hurson, AR; Desai, V.; Spink, A. and Miller, LL. (2001) "Transaction Management in a Mobile Data Access System", Annual Review of Scalable Computing, 2001

[Shi79] Shields, M. W.; Adequate Path Expressions. In *Proc. Semantics for Concurrent Computation*, Lecture Notes in Computer Science, Vol 70, pp. 249-265, Springer Verlag, 1979.

[Shi85] Shields, M. W.; Concurrent Machines. *Computer Journal*, 28:449-465, 1985.

[Shi88] Shields, M. W.; Behavioural Presentations. In *Proc. Linear Time, Branching Time and Partial Orders in Logics and Models of Concurrency*, Lecture Notes in Computer Science, Vol. 354, pp. 671-689, Springer Verlag, 1988.

[Shi97] Shields, M. W.; *Semantics of Parallelism*. Springer-Verlag, London, 1997.

[ShM04] Shields, M. W. and Moschoyiannis, S.; An Automata-Theoretic View of Software Components. Technical Report SCOMP-TC-02-04, Department of Computing, University of Surrey, 2004. Available at: http://www.computing.surrey.ac.uk/personal/st/S.Moschoyiannis/

[Shi05] Shields, M. W.; An Automata Theory of Components and their Composition. Technical Report SCOMP-TC-02-05, Department of Computing, University of Surrey, 2005.

[Sin97] Singh, M.; A Customizable Coordination Service for Autonomous Agents. In *Proc. 4th Int'l Workshop on Intelligent Agents – Agent Theories, Architectures and Languages*, Lecture Notes in Computer Science, Vol. 1365, pp. 93-106, Springer-Verlag, 1997.

[Smi99] Smith, J.R. (1999), Distributing Identity, *Robotics & Automation Magazine*, IEEE Publication, pp: 49 - 56 - ISSN: 1070-9932

[Sta03] Stallings W. (2003), Cryptography and Network Security Principle and Practice (Third edition), Prentice Hall, USA

[Tan89] Tanenbaum, A.S. (1989), Computer Networks, Engleweood Cliffs, NJ: Prentice-Hall.

[TeI03] Terai, K. and IzumI, N. (2003), "Coordinating Web Services based on business models", Proceedings of the 5th international conference on Electronic commerce (ACM Press), 2003, vol. 50, pp:473-478

[TUH+97] Tada, H.; Uchida, K.; Higuchi, M.; Fujii, M. (1997), "28 A model of nested transaction with fine granularity of concurrency control", 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Volume: 2 , 20-22 Aug. 1997, pp: 977 - 980 vol.2

[USD07] US Department of Health and Human Services, 'Fact Sheet: Isolation and Quarantine', Department of Health and Human Services, Centers for Disease Control and Prevention. Last accessed: Feb 2007. (http://www.cdc.gov/ncidod/dq/isolationquarantine.htm last access: 08/03/2007)

[VaP02] Valencia A. and Pazos F., "Computational methods for the prediction of protein interactions", Current Opinion in Structural Biology 2002, 12:368–373

[vdMDD+03] van der Meer, D. Datta, A. Dutta, K. Ramamritham, K. Navathe, S.B. (2003), "Mobile user recovery in the context of Internet transactions", IEEE Transactions on Mobile Computing, Volume: 2, Issue: 2, April-June 2003, pp: 132 – 146

[VeP98] Verharen, E. M. and Papazoglou, M.P.(1998), "Introducing Contracting in Distributed Transactional Workflows", Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7, 1998, pp:324-334

[VZG+05] Vogt, F.H.; Zambrovski, S.; Gruschko, B.; Furniss, P. and Green, A. (2005), "Implementing Web service protocols in SOA: WS-Coordination and WS-BusinessActivity", Seventh IEEE International Conference on E-Commerce Technology Workshops, 19 July 2005, pp: 21- 26

[Wan96] Wanlei Zhou (1996), "Performance evaluation of nested transactions on locally distributed database systems", Second International Symposium on Parallel Architectures, Algorithms, and Networks (IEEE CNF), 12-14 June 1996, pp: 353 – 356

[WaS06] Y. Wang and M. Singh. Trust Representation and Aggregation in a Distributed Agent System. Proceedings of *21st Conference on Artificial Intelligence (AAAI'06)*, pp. 1425-1430, 2006.

[WaS07] Y. Wang and M. Singh. Formal trust Model for Multiagent Systems. IJCAI'07, pp. 1551-1556, 2007.

[WaS98] Watts D. J., Strogatz S. H. "Collective dynamics of small-world networks", *Nature* 363: 202-204, 1998.

[Wat99] Watanabe, T (1999), "Agent-oriented model for managing long-lived transaction, based on work-flow and task-graph", Third International Conference on Computational Intelligence and Multimedia Applications ICCIMA '99 (IEEE CNF), 23-26 Sept. 1999, pp: 10 – 13

[Wik06] Wikipedia, (2006), "Web service", Wikipedia encyclopedia, Available at: http://en.wikipedia.org/wiki/Web_service (accessed date; 14/06/06)

[WiN95] Winskel, G.; Nielsen, M.; Models of Concurrency. In S. Abramsky, D. Gabbay, T. Maibaum, eds, *Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling*, pp. 1-148, Oxford Science Publications, 1995.

[WWZ02] Wgrzyn, S.W. Winiarczyk, R. Znamirowski, L. (2002). "Nanotechnologies and nanosystems of informatics as a basis for self-replication", IEEE-NANO 2002. Proceedings of the 2002 2nd IEEE Conference on Nanotechnology, Aug. 2002, pp: 99 – 102

[WZB+01] Xiao Weijun; Lu Zhengding; Li Bing; Sarem, M. (2001) "Transaction management in mobile multidatabase systems", 2001 International Conference on Computer Networks and Mobile Computing (IEEE CNF), 16-19 Oct. 2001, Los Alamitos, CA USA, pp: 513 – 518

[YaG02] Yang, B. and Garcia-Moline, H. (2002), "Designing a Super-Peer Network", Technical Report, Standford University, February 2002.

[YaM03] Yaohang Li; Mascagni, M. (2003), "Improving performance via computational replication on a large-scale computational grid", CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 12-15 May 2003, pp: 442 – 448

[YK-MA+02] Younas, M. Kuo-Ming Chao Anane, R. James, A. (2002), "Multi-agent transactions for Web-based design activities", The 7th International Conference on Computer Supported Cooperative Work in Design, 25-27 Sept. 2002, pp: 258 – 263

[YLY03] Yu, S. Zhou, W. Lan, M. Yue Wu (2003), "An architecture of Internet based data processing based on multicast and anycast protocols", PDCAT'2003. Proceedings of the Fourth International Conference on Parallel and Distributed Computing, 27-29 Aug. 2003, pp: 104 – 110

[YPH02] Yang, Jian; Papazoglou, M P. and Heuvel, W-J van den (2002), “Tackling the Challenges of Service Composition in E-Marketplaces”, Twelfth International Workshop on Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems, 2002. RIDE-2EC 2002 (IEEE Computer society), pp:125-133

[YSS04] Yu B, Signh M. and Sycara K. Developing Trust in Large-Scale Peer-to-Peer Systems. Proceedings of *First IEEE Symposium on Multi-Agent Security and Survivability*. IEEE, pp. 1-10, 2004.

[ZLB04] Zirpins, C.; Lamersdorf, W. and Baier, T. (2004), "Flexible coordination of service interaction patterns", Proceedings of the 2nd international conference on Service oriented computing (ACM Press), 2004, pp: 49-56

[ZoJ98] Zou H. Jahanian, F. (1998), “Optimization of a real-time primary-backup replication service”, Seventeenth IEEE Symposium on Reliable Distributed Systems, 20-23 Oct. 1998, pp: 177 - 185
Available from: http://users.ecs.soton.ac.uk/mjb [Last accessed: 19/07/2007]