Deliberate Cooperation In Service-Oriented Environments:
Dynamic Transactional Workflows For Web Services

A thesis submitted to the
University of Newcastle, NSW, Australia
for the degree of
Doctor of Philosophy
(Computer Science)

by

David John Paul
BMath, BCompSc (Hons I)

March, 2012
Statement of Originality

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. I give consent to this copy of my thesis, when deposited in the University Library, being made available for loan and photocopying subject to the provisions of the Copyright Act 1968.

(David John Paul)
Dedicated to my mother. You showed me the path but let me make my own way.
Acknowledgements

No man is an island entire of itself; every man is a piece of the continent, a part of the main.

*John Donne*

The city of Newcastle has a lot to offer. Large enough to have most modern conveniences available, but small enough not to have completely lost its village feel, the city combines the coastal lifestyle (including some of the most beautiful beaches in the world) with the industrial contrast that its mining history ensures. It has been my home; I can offer it no greater compliment or thanks. Of course, a location on its own, no matter how beautiful, cannot provide the support necessary for a long-term project such as this dissertation. For that, people are needed, and I’ve been fortunate enough to have had the guidance, support, and presence of some truly wonderful people, all of whom deserve my acknowledgement and my thanks.

Prominently among these people are my supervisors, Assoc. Prof. Frans Henskens and Dr Michael Hannaford. Their advice, encouragement, and, most of all, friendship, have proved invaluable. I must also thank everybody who has been involved in the Distributed Computing Research Group while I’ve been here: Mr Ali Alharbi; Mr Bilal Alguni; Dr Monjurul Alom; Ms Mukta Aphale; Mr Aaron Hector; Dr Bayu Hendradjaya; Mr Zainul Masud; Dr Aslam Nasir; Mr Vinh Nguyen; Dr Kim Nolan; Dr Peter Shaw; and Mr Mark Wallis. Especially Mark; your enthusiasm and seemingly undying ability to start work on new interesting problems, no matter how many other things you have going on, has been truly inspirational.

Outside of our research group, but still in the faculty, I owe a great deal to my confirmation committee: Dr Stephan Chalup; Dr Yuqing Lin; Dr Alexandre Mendes; and Dr Joe Ryan. The confirmation process was extremely positive for me, and this work would be much poorer without the excellent advice given after my first year of
postgraduate study. I’m also thankful for the excellent support given by the administrative staff, especially Ms Kathy Allan and Ms Kathy Killen, and the IT support from Mr Geoff Martin, Mr David Montgomery, and Mr Aaron Scott. I also wish to thank the Australian Schizophrenia Research Bank for my employment throughout my candidature, in particular Mr Jason Bridge, Dr Carmel Loughland, and Ms Kathryn McCabe.

In recent years, people have been realising the importance of a work/life balance; external “distractions” are actually beneficial and make a person more complete. For providing such a distraction, I thank Hunteract, the Rotaract Club of the Hunter, and, more recently, the Kiwanis Club of East Maitland. As I’ve already said, Newcastle still has a very village-like feel to it, and being involved in community service has made me feel worthy to belong; introducing me to some wonderful people was an added bonus. Exercise is equally important for a well-balanced life, and for that I must thank the Ragtags OzTag team and the Rainbow Sneakers netball team. I also thank some others who have made my life easier during the last few years (with “few” having quite a range for some of you): Ms Crystal Debreceny, Ms Nancy Earl, Prof. Maree Gleeson, Mr Geoff Keech, Ms Melanie Patfield, Ms Anastasia Suchowerska, and the Tomkins family (Nathan, Natalie, Gracie, James, and now Henry).

Finally, and most importantly, I offer my undying gratitude to my family, especially my parents John and Helen, my sister Catherine, and my grandmother June. You may not completely understand the process I’ve been going through, Dad, but you give your love and support anyway. You’ve had a much better idea Mum, encouraging me along the right path even when it may have seemed that I had lost my way. Catherine, your brilliance and perseverance have been of great influence, and I’ll always love you. And Grandma, if I can explain my work to you then I can explain it to almost anybody; thank you for listening and taking the time to understand.
## Contents

1 Introduction  
1.1 Cooperation In Service-Oriented Systems . . . . . . . . . . . . . . . . . . . . 2  
1.2 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3  
1.3 Research Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5  
1.4 Problem Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7  
1.5 Research Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7  
1.6 Research Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8  
1.7 Structure Of The Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9  
1.8 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10  

2 Background Information  
2.1 Service-Oriented Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . 11  
2.1.1 Similar Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
2.1.2 Web Services . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15  
2.2 Transactions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20  
2.2.1 Ensuring ACID properties . . . . . . . . . . . . . . . . . . . . . . . . . . 21  
2.2.2 More Complex Transaction Models . . . . . . . . . . . . . . . . . . . . . 24  
2.2.3 Multidatabase Transactions . . . . . . . . . . . . . . . . . . . . . . . . . 26  
2.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31  

3 Service Composition In The Web Environment  
3.1 Requirements Of A Service Composition Platform . . . . . . . . . . . . . . . . . 33  
3.1.1 Service Discovery . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34  
3.1.2 Uniform Information Exchange Infrastructure . . . . . . . . . . . . . . . . 35  
3.1.3 Adaptiveness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
CONTENTS

5.2.2 Determining A Client’s Next Action ....................... 107
5.2.3 Reasoning With Incomplete Information .................. 115
5.3 Negotiating Transaction Levels ............................... 116
  5.3.1 A Provider Negotiation Strategy ....................... 117
  5.3.2 Client Negotiation Strategies ......................... 118
5.4 Summary ................................................. 118

6 Simulating The Dynamic Transaction System .............. 120
  6.1 The Web Services Transactions Simulator ................. 121
    6.1.1 Simulating Messages ............................... 123
    6.1.2 Simulating Timing .................................. 124
  6.2 Simulating Participants ................................ 125
    6.2.1 Simulating Providers ............................... 125
    6.2.2 Simulating Clients ................................. 127
  6.3 Input For The Simulator ................................ 129
  6.4 Interpreting The Simulator’s Output ..................... 130
  6.5 Summary ............................................... 130

7 Validation ................................................ 132
  7.1 The Need For Non-Traditional Transaction Schemes ........ 132
    7.1.1 Scenario Description ............................... 133
    7.1.2 Results ............................................. 134
  7.2 Supporting Dynamic Transaction Levels ................... 136
    7.2.1 Scenario Description ............................... 137
    7.2.2 Results ............................................. 139
  7.3 Summary ............................................... 143

8 Conclusion ............................................... 145
  8.1 Research Questions ..................................... 146
    8.1.1 Research Question 1 ................................ 146
    8.1.2 Research Question 2 ................................ 147
    8.1.3 Research Question 3 ................................ 148
    8.1.4 Research Question 4 ................................ 149
Abstract

The only thing that will redeem mankind is cooperation.

*Bertrand Russell*

Modern society is complex and, in order to achieve desired goals, individuals must work together. Cooperation between parties can either be accidental, forced, or deliberate. Deliberate cooperation occurs when individuals realise that a successful outcome is more likely when they team up with others to achieve common goals. This thesis presents a method to support deliberate cooperation in service-oriented architectures. In such an environment, deliberate cooperation can be provided through improved transaction support.

Service-oriented architectures are based on the concept of services. Providers advertise the services that they offer and clients send requests for the services to be performed without needing to understand the intricate details of how the outcomes are achieved. Clients often require services from multiple unrelated providers in order to achieve their goals, but current systems make it difficult to combine these services in such a way that the client is guaranteed an acceptable outcome. Further, the existing standards are not always flexible enough to allow service providers to always offer their desired level of transaction support.

This thesis presents a method that allows service providers to dynamically alter the level of transaction support they offer for their services. This approach is more flexible than current approaches for Web Services transactions, and ensures that providers are always able to offer a level of support for cooperation with which they are comfortable. A formal system is also presented that allows clients to use the transactional guarantees offered by providers to reason about service compositions and ensure that client workflows always end in an acceptable state.
To augment these theoretical results, a Web Services transactions simulator has been developed. By simulating transaction flow rather than service flow, this allows the dynamic transaction scheme described in this thesis to be compared with more traditional Web Services transactions. Results indicate that support for dynamic transactional workflows can provide an overall benefit for both clients and service providers, and the simulator allows detailed study of how changes to the transactional behaviour of participants affects the outcome of particular scenarios.
Chapter 1

Introduction

Great discoveries and improvements invariably involve the cooperation of many minds.

*Alexander Graham Bell*

The last person classified as knowing everything\(^1\) was Thomas Young, who died in 1829 [104]. Since then, human knowledge has grown so quickly that nobody is able to keep up. It has been said that a single Sunday Edition of the New York Times has more information in it than an average person in the middle ages would learn in his or her lifetime [87]. New discoveries are constantly being made and “information overload” can drive a person to distraction. However, life goes on. Rather than being intimidated by the inability of an individual to be an expert in every field, humankind has learned how to specialise. The field of science has been broken up into individual branches such as biology, chemistry, and physics. These branches have been further split; physics, for example, includes electromagnetism, thermodynamics, and relativity, among others. Even these subbranches are too broad. The splitting is continued until it reduces to a manageable size, with areas such as classic electrodynamics or electrostatics.

While specialisation can help a person to not feel intimidated by the amount of information in the world, isolation does not. Without contact with people in other fields, any discoveries are, at best, limited. Great benefits can be introduced by combining knowledge from different fields. For example, bioinformatics applies statistics and computer science to problems in molecular biology, and has greatly increased understanding of the various systems of a cell [86].

---

\(^1\)Here, “knowing everything” means being familiar with practically all Western academic knowledge up to that point.
Thus, specialisation is necessary, but so is the sharing of those specialised skills and knowledge with other groups. Further, there is a need for people from different specialisations to work together, as often the required results benefit from expertise from different areas. To move from a purely scientific description to a real-world example, Habitat for Humanity New Zealand built a four-bedroom house in under four hours [60]. This would not have been possible without the various carpenters, electricians, plumbers, and other volunteers working together.

Cooperation is the process of working together to strive towards a common goal or outcome. Cooperation can be accidental, deliberate, or forced. Accidental cooperation occurs when, by aiming for their own goals, two parties help each other achieve a desired outcome. Forced cooperation is where a party has no choice but to work with another party to achieve a particular goal; this could be because of legal, monetary, or many other reasons. Deliberate cooperation requires two or more parties to realise that they have a common goal, that they are more likely to achieve it when they work together, and to thus do so.

1.1 Cooperation In Service-Oriented Systems

Service-oriented systems have decentralised parties that provide services, which can then be utilised by applications [96]. Each provider offers a set of services that it is willing to perform. Client implementations can then utilise services from multiple providers to achieve the client’s aims. As the client is free to choose services from any provider in the system, it makes sense for the client to always choose the provider that offers the best\(^2\) required service. Thus, if a provider is unable to offer a competitive service, then that service is unlikely to be utilised. This leads naturally to provider specialisation; a provider will only be called to perform services at which it can perform well.

Thus, each provider offers services that form the provider’s own specialisation, which are then utilised by a client to achieve that client’s aims. As services from different providers are being used to achieve a common goal, the providers can be seen as cooperating with each other. This cooperation is typically accidental; the providers take no special measures to allow the cooperation, it is simply the nature of the paradigm

\(^2\)Different clients may have a different idea as to what makes a particular provider better than another, but each client will have criteria that it deems important.
that allows the services to be combined. However, as will be seen in Chapter 3, the cooperation is sometimes forced.

*The contribution of this dissertation is to demonstrate a way that cooperation in service-oriented systems can be deliberate, and that this deliberate cooperation results in a better outcome for all parties involved.*

The remainder of this chapter is organised as follows: Section 1.2 motivates this research by examining how providers can support cooperation in service-oriented systems. In Section 1.3 we specify the aim of our research, which is more formally stated in Section 1.4. The problem is decomposed into specific questions in Section 1.5, and the methodology for addressing these questions is specified in Section 1.6. Section 1.7 outlines the structure of the remainder of this dissertation. Section 1.8 concludes the chapter by briefly describing the contributions of this work.

### 1.2 Motivation

Clients have a need to combine multiple services in a service-oriented environment into a single workflow. The client sees all of the providers in the workflow as participants working on achieving the client’s task. The client calls a service by sending a message to a provider with a request to perform an action and then waiting for a response to determine if the action succeeded. Each service call is part of an overall process, and the task will only be complete when the required service calls all succeed. Typically, this will require services from multiple providers being used by the client to achieve a particular aim. The various providers used by the client do not know that their services are being combined with those of other providers, and need not even know that the other providers exist; all a provider knows is that it is being asked to perform a service.

Thus, from the provider’s point of view, any cooperation is purely accidental. The provider is offering a service without considering how that service helps the client achieve a particular aim other than that satisfied by the single service it is providing. In particular, once a client sends a service request to a provider, the provider will either succeed or fail to perform that service. While this is completely acceptable when considering the provider as an isolated entity, it is not necessarily adequate when looking from the client’s point of view, in which a single provider’s action may be only part of a larger task. Consider,
for example, a situation where a client wants to go to a play. To allow this, the client must book tickets and hire a taxi to get to the theatre. There is a provider that offers a service to sell tickets, and another provider that offers a service to hire a taxi. When a client attempts a booking, there are three possibilities: both actions complete successfully; both actions fail; or one of the actions succeeds while the other fails. When both actions succeed, the overall client action is successful; the client will have tickets and transport to be able to see the show. When both actions fail, the overall client action is not successful, but the client is no worse off than before starting the process; the client has neither tickets to see the show, nor transport to get there. However, if one of the actions succeeds and the other fails, then not only has the overall client action failed, but the client is worse off. If the tickets have been booked, but the client cannot hire a taxi, then the client cannot get to the theatre and thus cannot see the show, meaning that the money spent on the tickets is wasted. Similarly, if the client is unable to obtain tickets but does successfully hire a taxi then the client can get to the theatre, but will be unable to see the show, and will have to pay for the taxi booking.

One possible way to avoid such a situation would be to require providers such as the ticket seller to offer the ability to cancel an order. This would improve the provider’s level of cooperation with the client, because if the client is unable to book the taxi but has booked the tickets, the client can cancel the ticket booking and it will appear as if neither action succeeded. While this is better for the client, it is not necessarily better for the provider. If the provider has already issued tickets, then there is a cost associated with cancelling the booking. Further, before the client has cancelled the booking, the provider cannot offer the tickets to anybody else. Thus, if another client requests to buy tickets before the first client cancels, but this fails because there are no tickets available, and no client requests to buy the tickets after the cancellation, then the provider has lost a sale that it would have made had the original client never booked.

Thus, forcing a provider to offer a particular functionality to improve cooperation between multiple providers for the client is unfair. As the provider is offering the service, it should be up to the provider to decide on the level of support to offer. Then, rather than accidental or forced cooperation, the provider would be offering deliberate cooperation at a level with which it is comfortable. A client could then take into consideration the support a provider gives for cooperation before deciding to utilise a service offered by the provider.
CHAPTER 1. INTRODUCTION

This seems the best solution, as the provider will only ever offer a level with which it is happy, and the client will only ever choose to use providers offering an acceptable level of support.

While the nature of service-oriented environments makes accidental cooperation between providers a necessity, and technologies such as some discussed in Chapter 3 bring forced cooperation to providers, support for deliberate cooperation has not been sufficiently discussed in the literature. In particular, there is no framework that allows providers to dynamically advertise the level of support for cooperation that they wish to offer, and thus no ability for clients to combine these offers to achieve an acceptable (to the client) combination. The work presented in this thesis proposes a remedy to this situation. In the following sections we further explain how we approach support for deliberate cooperation in service-oriented environments.

1.3 Research Objectives

The focus of this research is on the use of deliberate cooperation in service-oriented environments. With deliberate cooperation, providers choose to offer support that allows better cooperation between providers participating in a client workflow. In particular, we do not consider forced cooperation, where the provider has no or limited choice as to how it supports cooperation, or accidental cooperation, where the provider makes no provision for cooperation.

Any system that allows such deliberate cooperation must know which levels of support providers can offer. A provider can then use this information to make an informed decision as to how it will support cooperation. Further, the provider should be able to react to the current state of the system and change its support as its situation changes. Thus, for each service call, the provider should be able to decide the level of cooperation support it wishes to provide for that particular call.

As clients will wish to use services from multiple providers, and those providers are individually choosing the level of support for cooperation they wish to offer, it is necessary that clients do not require all providers to offer the same level of support. To allow this, the client must know how the various guarantees given by the providers can be combined. By examining these combinations, the client can determine whether they are willing to
CHAPTER 1. INTRODUCTION

utilise the services as they are offered, whether alternatives should be investigated, or if the overall action should be cancelled.

From the client’s point of view, what is important is the risk involved with the combined actions. No matter which level of support for cooperation is offered, a failure may occur if the provider cannot successfully complete the requested service. While different cooperation levels can mitigate the consequences of such a failure, only the client can determine whether the overall outcome would be acceptable. If a potential outcome of the combined services is not acceptable then the client should not request the services. The level of risk afforded by a particular set of services, each service with a particular level of support for cooperation, can be assessed on the basis of a formal system of rules specifying how the different levels mitigate the risk for a particular service call.

Current technologies do not provide a system that allows providers to offer a dynamic level of support for cooperation. Further, there is no formal system that allows clients to reason about the level of risk associated with combining the different levels of support offered by providers in such a system. The aim of this component of the research is thus two-fold:

1. To provide and evaluate a system by which providers can dynamically choose how they reduce the risk to a client in case of failure of a service call.

2. To develop a formal system that allows clients to evaluate the risk of combining service calls in such a system.

A single service call can offer many different risks to a client. For example, if one part of a client’s overall aim requires the client to order a part from a provider, then when that action succeeds the client must pay for the part. If the provider offers no support to mitigate risk, and the other actions that were required for the client’s overall aim fail, then the client has not only to pay for the part, but also to receive it, which then requires either storage or disposal. There are many other potential risks for a service call, such as threats to security or anonymity, or the risk that some quality of the service may not meet a particular level. Due to the complexity of potential risks in the case of failed service calls, this research is limited to consider a single possible risk, which may vary in magnitude. In the case of the given example, this could mean it is assumed that storage or disposal
could be handled by paying an extra fee. The single possible risk then becomes money, with only the amount changing depending on the other actions in the client’s aim.

The client’s risk is not the only important measure of the system. It is also important that providers’ performances can be compared. Similarly to a client’s risk, the success of a provider can be multifaceted. Because of this complexity, we limit the metric to the provider’s utility, which is again a single possible variable that can vary in magnitude. The exact definition of utility depends on the service being provided, but can typically be thought of as the profit the provider makes, or the number of service calls it successfully completes.

1.4 Problem Definition

As described above, a formal system that allows clients to evaluate the risk of performing a combination of different services offered by different providers will allow clients to proceed only if the level of risk is deemed acceptable. While the best result for a client is, of course, to have their overall aim completed successfully, the client is guaranteed that, even in the case of failure, the outcome is acceptable. However, allowing such a system requires providers to change their behaviour. Thus, it is important to know that it is beneficial for the provider to help mitigate the client’s risk. One purpose of this investigation is thus to design a system that allows providers to help mitigate the risks for a client and determine the following:

Problem Definition: Can deliberately reducing client risk increase a provider’s utility?

1.5 Research Questions

To answer the problem defined above, we decompose the problem into the following research questions:

1. *How has cooperation been achieved in the past and in current systems?* What are the current techniques and practises, and what can we learn from them?

2. *What levels of support can be offered to facilitate cooperation between multiple services?* What patterns can a service provider support, in addition to performing their actual service, to help clients achieve their goals?
3. *How can providers choose the level of support for cooperation they wish to offer?* 

Given the possible patterns, how can a provider choose and advertise the subset of patterns it wishes to support?

4. *How can the different levels of support offered by service providers be combined?*

If the different providers that a client wishes to use are offering different levels of support for cooperation, are they compatible, and what guarantees do they give? In particular, can a client make risk-based judgements as to whether or not it should proceed?

5. *Does a system that allows providers to dynamically decide on the level of cooperation offered for a particular service, and clients to combine services with different guarantees, give better results than are currently achievable? How can the benefits of such a system be measured?* A prototype implementation is required to show that the theoretical results lead to a practical solution. Furthermore, how would such a solution be implemented using current standards and technologies?

### 1.6 Research Methodology

This section specifies the methodology used for the research presented in this thesis. The methodology consists of five iterative stages, with each affecting the others.

The first stage is to define the problem. Preliminary ideas are used as a guide to direct the research, and the results of these early investigations are used to further specify the problem to be solved. The previous sections of this chapter are the result of this stage.

Moving on from the problem definition, the second stage is a search of existing literature in the field and related areas. This further defines the scope of the problem and allows open issues to be discovered. Chapters 2 and 3 examine existing knowledge and techniques.

The third stage is the design of a solution. Discussed in Chapters 4 and 5, this involves the development of a model that allows providers to dynamically choose the level of client support they wish to provide.

The next stage verifies that the solution does solve the problem, and evaluates how well it does so. This is described on a formal level in Chapter 5, and on a more practical level in Chapters 6 and 7.
CHAPTER 1. INTRODUCTION

The final stage is an appraisal of the proposed solution, looking at its benefits and shortcomings. This is achieved in Chapter 8.

1.7 Structure Of The Thesis

This section describes the structure of the main document:

Chapter 2 gives background information that is necessary to understand cooperation in a service-oriented environment. In particular, this chapter gives an overview of Web Services and transactions in general, specifying how each work and why they are used.

Chapter 3 introduces previous research into service composition in service-oriented environments, and specifically Web Services. This includes an overview of what is required for service composition, where and how Web Services transactions are useful, and the concepts of orchestration and choreography.

Chapter 4 presents an approach that allows more flexibility for providers to offer transaction-like support in a service-oriented environment. Information is given on the different levels of support that can be offered and an overview of how they can be combined. Finally, a mechanism is described for including such a system in the current Web Services environment.

Chapter 5 extends the previous chapter by giving a formal basis to the suggested system. This is then used to examine a client’s use of such a system and how risk can be calculated.

Chapter 6 describes a simulator for the model presented in the previous chapters. This simulator models transaction flow rather than service flow to provide a fair environment to compare various transaction schemes.

Chapter 7 validates the simulator and the main ideas of this thesis by describing experiments that verify the functionality of the simulator, and the viability of a dynamic transaction scheme.

Chapter 8 concludes the work, and discusses future avenues of research.
1.8 Contributions

This thesis presents a new approach to supporting cooperation in a service-oriented environment. By examining how cooperation is currently achieved, weaknesses and limitations of the existing systems are identified and a proposal for their improvement is put forward. This is achieved by developing a theory and model to allow different levels of cooperation support to be dynamically selected and combined as the system evolves. This model has been developed into a prototype and a description of how the model could be integrated with current technologies. The prototype has been evaluated to show the effectiveness of the new model.

Publications arising from work completed for this thesis are as follows:

“Isolation and Web Services Transactions” [98] introduces the concept of provider callbacks, described in Chapters 2 and 4.

“Transaction Support for Interactive Web Applications” [101] describes concepts of transactionality in the Web and similar environments, which are explained in more detail in Chapter 3.

“Web Browser Transactionality” [117] further details transactional concepts in the Web environment and describes the inclusion of client-side transactional support in a Web browser.

“Per-request Contracts for Web Services Transactions” [99] introduces the concept of dynamic contracts for transactions involving Web Services, the ideas of which are expanded in Chapters 4 and 5.

“Simulating Web Services Transactions” [100] provides an overview of the simulator presented in Chapter 6 and presents results of a validation experiment described in Chapter 7.
Chapter 2

Background Information

Where there is a will there is a way. And this must be the way not of compulsion but of cooperation. No government and no plan can succeed without it.

Lionel K. Murphy

In order to understand the current state of cooperation in service-oriented architectures, it is first necessary to examine the problems that service-oriented architectures have been designed to solve, and how such systems are currently implemented. Section 2.1 introduces Web Services, the most commonly used service-oriented system [96], by first looking at related earlier technologies and then describing how Web Services attempt to solve some of the problems found in the earlier systems. As well as an understanding of service-oriented architectures, it is also beneficial to look at how cooperation has been achieved in other distributed systems. To this end, Section 2.2 describes how transactions can be used to combine multiple actions into a single logical action. Further, this section explains how traditional transactions in a single database system have been modified to allow their use in multidatabase systems. The information presented in this chapter will be beneficial when Chapter 3 introduces the concept of Web Services transactions.

2.1 Service-Oriented Systems

This section introduces the concept of a Service-Oriented Architecture (SOA). SOAs allow multiple systems to communicate with each other to perform required tasks. An SOA is founded on the use of basic services, descriptions of the services, and basic operations such as publishing, selecting, and binding to services. These simple foundations allow the
creation of useful software by combining the simple services to perform complex behaviours [96].

One of the main features of SOAs is that they are loosely-coupled. Services can be used without understanding the low-level details of how they actually work; all that matters is the result of calling the service. Loose coupling makes it easy for a service provider to change their implementation of a service without affecting their clients. As long as the service continues giving the correct results, it does not matter how it achieves them. Similarly, it is simple for clients to replace a service that they have previously used with one that is, perhaps, offered by another company or performs better (that is, it may be guaranteed to be cheaper, or faster). Such a replacement does not necessitate the changing of the details on how services are combined; all that needs to change is which service is used. Thus, once an application that combines services has been defined, the actual services used by the application can be changed and the application will continue to work.

The remainder of this section is organised as follows: Section 2.1.1 examines previous systems that have similar goals to the service-oriented paradigm. An examination of these systems is useful to identify the issues that SOAs have been designed to overcome. Section 2.1.2 then introduces the most commonly used SOA, Web Services.

### 2.1.1 Similar Systems

Many different systems have been developed to facilitate communication between programs running on heterogeneous systems. The most notable attempts have been RPC, RMI, DCOM and .NET, and CORBA, all of which are briefly introduced in this section. While some of the presented techniques are more sophisticated than others, a basic knowledge of each is required to understand how such solutions have evolved, and the need for other solutions, such as Web Services.

#### 2.1.1.1 RPC

A Remote Procedure Call (RPC) allows a program executing on one host to call a subroutine executing on a remote host. The programmer incorporates the remote call as if it were a local call, and an RPC tool hides the complexity of sending the call, including its parameters, across the network and receiving a response [37]. Programs are created
by combining procedures, some of which are set to be executed remotely, and the RPC tools ensure that this occurs transparently to the developer of the program.

As programs in procedural languages are already broken up into procedures, RPC is a natural extension of these languages. Thus, developers can concentrate on actually solving a problem rather than the low-level details of how network communication is achieved. Once a solution is developed, the programmer then indicates which procedures should be called on a remote host, and the RPC tool creates stub program code that automatically uses the remote hosts to perform those operations.

2.1.1.2 Java RMI

Java’s Remote Method Invocation [110] allows programs to transparently communicate with objects over a network. Thus, RMI can be thought of as object-oriented RPC for Java. A client object is able to call methods of a server object, which then returns the results back to the client. Whereas RPC is a natural extension of procedural languages and allows specified procedures to run remotely, RMI extends Java so that entire objects, with their associated instance data and methods, can be accessed over the network. This allows programmers to solve problems using object-oriented techniques and then have these solutions execute using the resources of multiple systems.

2.1.1.3 DCOM and .Net

Microsoft’s Distributed Component Object Model (DCOM) [22] allows applications written on the Windows platform to communicate over networked computers. It is an extension of Microsoft’s Component Object Model (COM). As such, DCOM is language neutral, and implementations can be changed without other software components that use features in that implementation being aware that a change has occurred. DCOM makes extensive use of Microsoft Remote Procedure Call (MSRPC) [106] to allow the sending and receiving of data via the network. Thus, DCOM is built on a system similar to that described in Section 2.1.1.1.

While DCOM was once Microsoft’s main technology for distributed computing, it has now been deprecated by the .NET framework [78]. It is still possible to access COM components with .NET, and so DCOM can still be used, however .NET provides many enhancements to the older systems. For example, the .NET framework includes extensive
libraries that offer support to help solve problems, such as .NET Remoting [85] for network communication; .NET Remoting allows objects to be made available across remoting boundaries, which could be application domains, processes, or computer networks. Further, as with Java RMI, clients can directly access objects residing on a remote server, which can then be used as if they were local objects. One major difference between Java RMI and .Net Remoting is that the interfaces of remote objects are generated at compile time for RMI, and at run-time for .Net [42].

2.1.1.4 CORBA

Object Management Group’s Common Object Request Broker Architecture (CORBA) [94] is an object-oriented middleware that allows clients to access objects hosted on remote, possibly heterogeneous, systems. The underlying component of CORBA is the Object Request Broker (ORB), which allows objects to send and receive responses transparently over a computer network. The ORB is responsible for achieving this interoperability between objects possibly implemented in different environments. There is a set of standard Object Services that describe how to use and implement objects in the system; these provide details of, for example, object life cycles, how objects in the system are named, and the relationships between different objects. As well as these, there are also Common Facilities and Application Objects that extend the standard in ways that are either group or vendor specific. Once a programmer has registered an object with the ORB, it can be accessed through the ORB and used in a standard way, even if the accessing object resides on a remote host.

CORBA provides many features, making it one of the most comprehensive middleware systems for allowing a distributed set of services to be integrated into a single application. It can be used on hardware ranging from mainframes to handheld devices and can allow programs written in different languages on different architectures to easily communicate over a network. By separating the interface definition from the actual implementation, it is also possible to replace a program written in one language with a new, improved version, possibly written in another language, that provides the same interface. The improved version could then replace the old version without clients realising that the implementation has changed (except, perhaps, by observing the enhanced performance).
CHAPTER 2. BACKGROUND INFORMATION

2.1.1.5 Evaluation

The main problem with RPC, DCOM, .NET, and RMI is that they are not compatible with many systems or programming languages; RPC has many different implementations on many different architectures that are all incompatible; DCOM and .NET are really only supported on Microsoft systems\(^1\); and RMI only works with programs written in Java\(^2\). As such, these technologies cannot easily be used in a heterogeneous environment, where the hardware, operating system, and programming languages are potentially completely different for all connected systems.

CORBA does not suffer from these problems, but does have problems of its own [62]. Firstly, the specifications are very complex, somewhat ambiguous, and difficult to fully implement. Further, initially all CORBA services needed their own port, which had to be accessible through any firewalls, and all information was sent unencrypted. While it is now possible to use SSL and configure recent implementations to use a single port, these features are fairly new, meaning that the use and uptake of CORBA had declined greatly before such features were available. Finally, CORBA requires a deep understanding of all component systems before they can be combined. This understanding may require intimate knowledge of the internal structure of a corporation, which is often private information.

2.1.2 Web Services

A Web Service is a “software system designed to support interoperable machine-to-machine interaction over a network” [121]. Web Services allow programs written in practically any language, on all modern platforms, to expose their services through the commonly used HTTP (80) or HTTPS (443) ports. There is no need to understand how a service actually works; all that is needed is an understanding that the service will do what is specified. In this way, Web Services can overcome the main problems with the other systems. Programs, possibly written in different languages and executing on different hardware or operating systems, can then all access a Web Service in a standard way. Through the use of standard protocols, neither the client nor the server need know the differences between the systems with which they are communicating; the server receives a standard formatted request,

\(^1\)Though projects such as The Open Group's COMSource [112] and the Mono project [83] provide some support for other systems.

\(^2\)RMI-IIOP does allow RMI to be used over IIOP, as used by CORBA, however its interface is not entirely compatible with RMI-JRMP. Further, very few CORBA implementations support all the features required to allow interoperability with RMI-IIOP.
which it executes before sending back a properly formatted response.

### 2.1.2.1 Web Services Standards

Web Services are based on open standards. As a result they can be easily implemented on any new architecture or system for which they are required. Any systems that support these standards can then communicate with each other without regard for the systems’ differences. While there are many different standards used in the implementation and use of Web Services, the three main standards are SOAP [59], WSDL [34], and UDDI [35]. Other Web Services standards are used to provide extra features that are not included in the base three, for example to add security or reliable messaging to Web Services [71, 48].

#### 2.1.2.1.1 SOAP

SOAP [59] is a message protocol, used for sending XML-based messages. The use of XML is both an advantage and a disadvantage. As XML is quite popular, there are numerous parsers available for extracting information from SOAP messages, and it is also possible for the messages to be written or edited by a human. However, XML is quite verbose, so SOAP messages can take longer to send and process than the binary messages used in other middleware technologies, such as CORBA. The openness of the SOAP protocol, however, makes it inherently multi-platform, and allows it to be easily extended if necessary.

All SOAP messages are composed of two parts: the header, and the body. The header contains information useful to hosts that receive the message en route to its ultimate destination. For example, a deadline could be specified and any receiving host could use this information to prioritise routing. The body contains information intended for the final receiver only. This could be a request, or a response to a previous request.

SOAP messages are typically sent via HTTP or HTTPS, but are, in fact, protocol independent. The HTTP and HTTPS protocols, however, are so commonly available that their use allows the vast majority of hosts to communicate. Further, because the protocols are so ubiquitous, their use avoids many of the problems that other protocols have with proxies and firewalls. However, since SOAP is protocol independent, practically any protocol can be used, provided that all participants in a message exchange understand how the elements of the SOAP messages are bound to the protocol. Other published protocols for sending SOAP messages include SMTP [39, 84] or UDP [36].
2.1.2.1.2 WSDL

The Web Services Description Language (WSDL) [34], as the name suggests, is a language for describing Web Services. Similarly to SOAP, WSDL is XML-based, with similar associated advantages and disadvantages. However, as a WSDL document simply describes a set of Web Services, and each service rarely changes, the WSDL document only needs to be sent to a client once and the described Web Services can then be used multiple times. Thus the increased verbosity of the XML needed for WSDL documents is not as great a disadvantage as it may first seem.

WSDL describes a Web Service as a collection of endpoints, called ports, that operate on messages described in the WSDL document. Thus, ports are associated with network addresses, and messages describe the data transferred between these ports to supply the service. Both ports and messages are defined abstractly, so definitions can be reused many times throughout the document. Thus, WSDL allows an implementation-neutral description of the services offered by a Web Services provider. Clients process the information in a WSDL file to unambiguously discover how to request services from a particular provider. The client can then use the protocol bound in the WSDL document to access the services offered by the provider. In the current Web Services environment, the specified protocol is nearly always SOAP, though WSDL does not require this.

2.1.2.1.3 UDDI

Universal Description, Discovery and Integration [35] is an advertising and discovery protocol, providing a registry that service providers can use to list the services they provide, and which consumers can search to find a provider of a required service. It is designed so that SOAP messages can be used to query the registry, which then returns WSDL documents describing how to interact with a service provider that matches the given criteria. UDDI is broken up into three components: the white pages, the yellow pages, and the green pages [114]. The white and yellow pages are analogous to telephone listings; the white pages allow a search by known identifiers, such as name or address, and the yellow pages categorise different services and providers based on what is done by the offered services. Green pages give technical information about the services offered by a particular provider, for example references to various Web Services specifications or file- and URL-based discovery mechanisms.
One of the main problems with UDDI is that there is no specified way to categorise services [12]. Thus, different registries may categorise services in disparate ways. As a consequence, a search that is successful on one registry may be unsuccessful on another, even if both list the same services. This problem, and the fact that there are currently relatively few service providers offering competing Web Services, has led to poor usage of UDDI. Instead, clients somehow independently discover which service provider they wish to use and contact it to obtain a description of the services offered. As Web Services become more common and registries become more necessary, there will hopefully be a standard definition of categories so that searches are not registry dependant. This will greatly increase the usefulness of UDDI by making Web Services much easier to find.

Another problem with UDDI is that it is a centralised service [111]. While replication can alleviate the single point of failure problem, ensuring that all replicas are up to date is non-trivial. There have been suggestions to create a peer-to-peer protocol for setting up Web Services registries, such as [12, 111], and, while none of these have become very popular, they do remove the problem of centralisation. Further, if standard ontologies are created for Web Services in particular industries, then these peer-to-peer registries could easily be combined with existing UDDI registries to greatly improve the services offered individually by either registry.

2.1.2.1.4 Other Web Services Standards

While SOAP, WSDL, and UDDI are the main Web Services standards, many other Web Services standards have been published. Most of these standards provide useful functionality, however many of them compete with other standards offering similar functionality. This makes it difficult to determine which standard should be used for a particular circumstance. A sample list of other Web Services standards are:

- Web Services Addressing (WS-Addressing) [18]: Provides protocol-independent addressing information. Thus, instead of relying on network-level transport to describe routing information as well as conveying the message, the addressing information can be added to the SOAP header, meaning that the transport protocol’s only responsibility is to deliver the message.

- Web Services Security (WS-Security) [71]: Enables use of XML encryption and
signatures for SOAP messages. This allows the integrity and confidentiality of SOAP messages to be guaranteed even when the message is transported over an insecure protocol.

- Web Services Reliable Messaging (WS-ReliableMessaging) [90]: Provides reliable communication for messages. This allows guarantees about how messages are received, such as ensuring that messages are delivered at least once or in a particular order, even when sent over unreliable infrastructure.

- WS-Callback protocol (WS-Callback) [53]: Allows asynchronous communication via Web Services. This allows asynchronous responses to a SOAP request to be sent to a destination that is not known at deployment time.

- Web Services Notifications (WS-Notification) [54, 116, 115]: Provides a publish-subscribe system for sending messages. This allows a Web Service to send notifications of particular events, without it having to know where to send those notifications before deployment.

- Web Services for Application Session Services (WS-Session) [41]: Allows applications to create and maintain a relationship with servers. This allows data to be retained between service calls, so that later invocations remember the previous activity.

- Web Services Resource Framework (WSRF) [13]: Adds ability for Web Services to be stateful. This allows clients to use resource services to store and retrieve data, and send a reference to this data with the Web Service request. This is more flexible than WS-Session, because it can use WS-Addressing to specify the location of the resource service.

- Web Services Agreement (WS-Agreement) [7]: Establishes agreement between two parties. The standard provides an abstract mechanism through which two parties can provide or obtain guarantees related to the services that are being used.

- Web Services Business Process Execution Language (WS-BPEL) [88]: Specifies business process behaviour. This orchestrates multiple Web Services interactions into a single process, including logic such as whether particular calls should be made, and how errors should be handled.
CHAPTER 2. BACKGROUND INFORMATION

- Web Services Choreography Description Language (WS-CDL) [122]: Describes peer-to-peer collaborations of participants. Whereas WS-BPEL specifies a business activity for a single entity, WS-CDL describes the observable behaviour of a process being executed by multiple independent participants.

2.1.2.1.5 Web Services As A Service-Oriented Architecture

While Web Services can be used in a similar way to RPC, the real power of the platform comes when it is used to implement an SOA. This is achieved by using the Web Services standards to provide the basic requirements of a service-oriented system. Each Web Service provides a basic service. These are described by WSDL, which gives an unambiguous description of how the services can be accessed. UDDI then allows the publishing of services, as well as the ability for clients to select a required service. Finally, SOAP can be used by a client to bind to a particular service.

2.2 Transactions

An important consequence of the simple services used in SOAs is that it is possible for a client to combine services, possibly from different providers, into a single, value-added service set. For example, if a printing service only accepts a particular document format, a client may use a different format with the service by first utilising a conversion service to ensure that the correct document type is passed to the printing service. Such combinations of services can lead to difficulties, especially when one or more of the combined services fail. As far as the client is concerned, the combination of services should be seen to provide a single facility; for example, the client does not need or want to convert the document if the service that requires the different format fails. However, as currently described, services do not allow such combinations to be seen as a single action; each service is offered completely independently of all others.

Transactions, which combine multiple actions into a single logical action, are a possible solution to this problem. Frequently used in database systems, transactions ensure various properties to guarantee correct behaviour, even when errors occur. This section examines some of the concepts underlying the use of transactions in database systems, with the view of using similar techniques to help control the outcome when multiple services are
combined in an SOA.

2.2.1 Ensuring ACID properties

Transactions typically provide the ACID properties of Atomicity (ensuring that all actions in the transaction either complete successfully, or revert to a state where none of them occurred), Consistency (ensuring that the system is not put into an illegal state), Isolation (letting concurrent transactions execute as if they were the only transaction being run, and ensuring that they do not interfere with each other), and Durability (ensuring that any completed transaction has its outcome recorded on a stable medium and cannot be undone, even by accidental events such as hardware or software failure) \[56\]. By ensuring these properties, transactions are guaranteed to complete in a predictable way, making it impossible for the transactions to perform operations that would compromise the integrity of the system. Special measures are needed to ensure that the ACID properties are upheld, especially when multiple transactions can run concurrently \[55\].

2.2.1.1 Concurrency Control

The simplest method to ensure the isolation property is to only allow a single transaction to run at any one time. The transaction then has complete control, and is guaranteed that no other transaction will interfere with any of the resources that it requires. However, most transactions only require access to part of the database. Blocking access to resources that are not affected by the currently-executing transaction would be impractical. Instead, it should be possible for transactions to be executed concurrently. However, this does require special care to ensure that the isolation property is not violated.

The main problems caused by violation of the isolation property are lost updates and dirty reads. A lost update occurs when concurrent transactions incorrectly update the same value. For example, if a bank account has a starting balance of $100 and two transactions read this value before attempting to deposit an extra $50, the first transaction would correctly set the balance to $150, however the second transaction would also set the balance to $150. Thus, the first transaction’s deposit would be lost, because the second transaction was not aware of the first transaction’s activity when it made its change. A dirty read occurs when a transaction reads a value updated by another transaction that subsequently fails. For example, if a transaction updates a bank balance to $150, another
transaction reads this value, and then the first transaction fails, returning the balance to $100, the second transaction should not continue assuming that the balance is $150.

Thus, for performance and usability reasons, it is preferable to allow multiple transactions to execute concurrently, but there are problems that must be overcome. Different forms of concurrency control can be used to ensure that these issues are avoided while still allowing concurrent access. The two main categories for concurrency control are pessimistic and optimistic. Both ensure that isolation is maintained, but do so in different ways. As such, each technique is better suited for different types of database access, and each has a performance penalty that increases when used in the inappropriate situation.

2.2.1.1 Pessimistic Concurrency Control

Pessimistic concurrency control relies on locks to ensure that two actions do not use data in a way that would leave the database in an inconsistent state [56]. Whenever a transaction wishes to use a database object, it must first obtain a lock for that object, appropriate for the kind of operations it needs to perform. If another transaction subsequently requests an incompatible lock, it will be forced to wait until the first transaction releases its lock. There are typically at least two kinds of locks: read (or shared) locks and write (or exclusive) locks. A read lock allows the transaction to read the object from the database, but not change it. A write lock indicates that the transaction plans to perform some action that will transform the database object in some way. Multiple transactions can then hold read locks on the same database object as, since none of them change the value of the object, their operation will not affect the other transactions. However, if at least one transaction has a read lock on a database object, then no transaction can obtain a write lock on that object, as that could possibly result in a dirty read; any transaction requiring a write lock would need to wait until all read locks had been released. Similarly, if a transaction has a write lock, then no other transaction can obtain a read lock for that database object. Further, to avoid lost updates, no other transaction can obtain a write lock on the database object until the current write lock has been released.

Thus, when using a pessimistic concurrency control scheme, it is possible that transactions will spend a large amount of time waiting until they are able to obtain locks on the objects they need to use. In fact, it is possible that two (or more) transactions
may be holding locks for some resources and waiting for other transactions to release locks for other resources it also needs. When two or more transactions are blocked, waiting for the others to complete, they are said to be deadlocked [55]. None of them are able to perform their work until the others finish, meaning that they would wait forever. Typically this situation is resolved by detecting when a deadlock occurs, and forcing transactions to abort until the deadlock is broken. However, even if transactions are guaranteed not to remain in a deadlocked state, the checks to determine whether locks should be granted can still take a significant amount of time. This time is largely wasted whenever it is extremely unlikely that multiple transactions would interfere with each other, such as in a large database with access spread over the entire store. In such cases, optimistic concurrency control techniques are often the better choice.

2.2.1.1.2 Optimistic Concurrency Control

Optimistic concurrency control allows transactions to perform whatever actions they need to perform, but detects whenever multiple transactions access objects in such a way that a breach of isolation could occur. When such a problem is detected, the system forces one or more of the transactions that caused the problem to be rolled back and sets them to start again at a future time. The ability to concurrently access and mutate data is typically achieved by providing each transaction with its own copy of the relevant parts of the database. Each transaction then performs any changes on its copy and then, before writing any changes back to the actual database, using a validation phase to determine if any other transactions would conflict with these changes. The validation phase can either check for conflicts between just the committing transaction and the transactions that have already been committed, or it can consider all currently-running transactions. Whenever a conflict is detected, some of the transactions causing the conflict are rolled back, and rescheduled, until the set of executing transactions no longer causes a conflict. When many collisions occur, the cost of constantly restarting transactions can be significant. However, when conflicts are rare, validation can be performed quickly and the optimistic techniques can have better performance than the pessimistic ones.

Systems using optimistic concurrency control can suffer from livelock, a similar condition to the deadlock in pessimistic systems. This occurs when a (typically long-running) transaction is continually cancelled and restarted because other, faster
transactions are changing data used by the longer-running transaction, and being committed before the other transaction can finish. In essence, the longer-running transaction cannot finish because it is always being interrupted by the shorter transactions. However, similarly to deadlock, this situation can be detected, and thus avoided.

2.2.1.1.3 Hybrid Concurrency Control

While pessimistic and optimistic are the broad categories for concurrency control, there are hybrid systems that do not neatly fit in either category. The simplest of these merely allows manually changing from one sort of concurrency control to the other. Other, more sophisticated, systems can automatically choose which to use based on historical or other data stored by the system [21]. Finally, there are also techniques that perform as well as the best of either pessimistic or optimistic concurrency control in any situation. One such technique is dependency graph-based transaction and concurrency control (DCC) [10, 11, 65, 64].

As the name indicates, DCC relies on dependency graphs, which are used to record inter-entity activity. Whenever a transaction uses data, a directed edge is added to a graph to indicate that data has been read or written (with the direction of the edge depending on how the data is accessed). As each edge is added, the graph can be checked to ensure that the recorded access does not cause problems such as lost updates or dirty reads. If there is a potential problem, then transactions can be rolled back until all such problems are avoided. Finding which transactions to roll back is achieved by traversing the edges of the dependency graph. By allowing, but tracking, all accesses to the data, it is possible for DCC to allow more transactions to complete successfully than systems that simply record which data was accessed without specifying how that data was used. DCC is also able to detect problems part-way through a transaction, allowing transactions to be aborted and restarted when the problem is first discovered.

2.2.2 More Complex Transaction Models

One problem with the transactions described so far is that even if a transaction fails at its very last action, the transaction must be started from the very beginning if it is attempted again. For simple transactions this is acceptable (and desirable), but as the work involved
before a transaction fails becomes more complicated, it would be better to avoid having to repeat all of the work if that can be avoided [56]. One method used to help achieve this is the use of savepoints. When a transaction has performed some work, a save point can be set up and any subsequent failures can revert back to that savepoint, from which actions can be performed to try to avoid subsequent errors.

A variation of savepoints is chained transactions. Each transaction in a chained transaction scheme is a sequence of subtransactions that are attempted in order, with a later subtransaction only beginning if the previous subtransaction successfully commits. In this way, a rollback of the current subtransaction is basically the same as reverting to a savepoint after the previous subtransaction had finished. This can be thought of as returning to the previous savepoint; in chained transactions there is no ability to rollback to any savepoint except the most recent. One of the advantages of chained transactions over savepoints is that, when a subtransaction is complete, it can release any locks obtained by that subtransaction, or, if optimistic concurrency control is used, the completed subtransaction is less likely to have used database objects used by other transactions, because it is only a part of the entire transaction, and has accessed a smaller number of objects.

Savepoints have also been generalised into nested transactions [43]. Savepoints organise a transaction into a sequence of actions, where one performs after another. Nested transactions, on the other hand, arrange the transaction’s actions in a hierarchy. Each subtransaction becomes a single action to the main transaction, which can either succeed or fail. Each subtransaction can consist of further nested subtransactions. If a subtransaction has to roll back then its parent transaction must also roll back. If a subtransaction commits, then its commit is not actually stored until its parent transaction commits. In this way, the original transaction is committed only if each of its subtransactions also commits. If any subtransactions fail, then so does the main transaction, which then rolls back all of its subtransactions. Subtransactions can often be run in parallel, which increases the efficiency of the overall transaction when compared to savepoints.

Multi-level transactions, or sagas [52], are a further generalisation of nested transactions. A saga is essentially a chained transaction in which each transaction in the chain has a compensating transaction. A compensating transaction undoes the work of a
successful transaction in the chained transaction, and is enacted whenever a later transaction in the saga fails. Thus, a saga either commits after each transaction in the chain successfully completes, or, by performing the compensating transactions for each already-committed transaction in the reverse order to which the transactions were originally run, the saga rolls back after the failure of one of the transactions in the chain. This keeps the atomicity of the transaction, but failure of a single transaction in the chain undoes the entire transaction; if the last transaction in the chain fails then all previous transactions’ compensating transactions effectively roll back the entire transaction chain. Sagas limit the time that a transaction holds resource locks, which can improve overall performance of the transaction system.

Finally, multi-level transactions can be extended into open nested transactions [118]. An open nested transaction is a multi-level transaction that also has the ability to start other top-level transactions without any further control. Thus, the newly started transaction may commit while the original open nested transaction fails, or it may fail despite the successful commit of the original transaction. Open nested transactions are thus more general than multi-level transactions, however they have less control over the actions they perform. This can lead to inconsistency, though the performance benefits of open nested transactions over nested transactions are often considered worthwhile [118].

2.2.3 Multidatabase Transactions

Transactions in single database systems are typically very short-lived. Thus, it is normally acceptable for either pessimistic or optimistic concurrency control techniques to be used, because the short time period limits the overhead of the concurrency control. For short transactions it is sometimes useful to provide even stronger guarantees than those afforded by the traditional ACID properties, as stronger ordering of transactions allows further optimisations [113]. On the other hand, when a transaction uses multiple heterogeneous databases, transactions take significantly longer to complete. Firstly, this is because of the communication delays necessary to access the multiple databases, which increases the length of the transaction to seconds. Further, the response time of the different systems may be increased because of issues such as geographic considerations. For example, if trying to trade stocks on both the Australian and New York exchanges, it may be necessary for the transaction to wait for hours, until one or both of those exchanges opens, before it
can continue processing. Other issues that affect the length of multidatabase transactions include the necessity of human interaction for some tasks, time taken for a physical act to occur, and systems that may take considerably longer than other systems to respond.

When combining services from multiple systems into a transaction, it is typical to use a two-phase commit protocol (2PC) [1]. This, or something similar, is required to ensure that the ACID properties are maintained. In a 2PC protocol, a transaction coordinator first sends a prepare message to each participant in the transaction. On receipt of such a message, a recipient determines whether it can perform the actions required of it for the transaction. If the recipient is willing to guarantee that it can complete the requested actions, it replies with a success message and does whatever is necessary to ensure the requested actions can be completed successfully. Thus, if the coordinator receives a success message from each participant, it is guaranteed that the transaction can succeed. The coordinator then sends a commit message to the participants, each of which must then perform the required actions. However, if a recipient of a prepare message is unable or unwilling to perform the requested actions, the coordinator is sent a fail message. In this case, the transaction cannot be completed and the coordinator must send an abort message to each participant, informing them that any guarantees they had provided are no longer required. In this way, no action is performed until the transaction coordinator is certain that the entire transaction will succeed (and thus sends the commit message), or, in the case that it is impossible for the transaction to complete, no action takes place (as the coordinator’s abort message stops any participant from performing their action).

One of the major problems with 2PC is that after a participant has sent a success message to the transaction coordinator, it must ensure that the requested action will complete successfully on the receipt of a commit message. For example, if a flight booking request sends back success then it must ensure that a seat is available on the flight. If another person attempts to book the same seat on the flight, then the system must wait until the first transaction completes before it can let the person know whether their request has succeeded or failed. The period of time that the system must wait before receiving either the commit or abort message is indeterminate, and could possibly even be forever if the connection between the system and the transaction coordinator is broken or removed.

When a provider’s resources cannot be used because of a transaction that may need them, those resources are said to be locked. Such locking of resources can significantly
reduce the entire system’s level of service, as later requests must wait for the transaction to complete before they can receive their results. Typically, in non-distributed situations, this is not a great problem, as transactions only have a short duration. However network delays and unreliable communication links already make transactions combining multiple systems execute significantly slower than transactions on a single system. Further, in multidatabase systems it is common to execute cross-institutional processes that require cooperation between multiple otherwise unrelated parties, and, as a result, transactions can run for even longer periods of time [76].

Another problem with locking resources in multidatabase systems is that each service provider would like to remain autonomous, allowing them to cancel a pending request whenever they choose. This is obviously not possible using 2PC, as, once a provider has guaranteed that an action can complete successfully, the provider must wait for the transaction coordinator’s message before being able to abort or commit the action. Thus, while technically possible, it is often considered impractical to have full transactional support in multidatabase systems. Individual providers are typically not willing to lock resources for the length of time needed, nor give up their autonomy to the degree required, for fully ACID transactions to be commonly used [95].

Despite the fact that fully ACID transactions are seen as unachievable for multidatabase transactions, it is possible to support transactions by slightly reducing some of the required properties. When considering which ACID properties can be reduced, consistency and durability seem to be required; the system should never be in an inconsistent state, and a system that unpredictably changed previously guaranteed decisions would be practically unusable. Thus, the most likely candidates for a reduction in strength are atomicity and isolation.

2.2.3.1 Relaxation Of Isolation

While it is sometimes possible to fully retain isolation in multidatabase transactions in a similar way to 2PC [1], the use of such a protocol reduces the autonomy of the individual participants in the multidatabase system. Once a system sends the success message, it is necessary to wait for a response from the coordinator before the system can abort or commit the required actions. This can be alleviated by using extensions to the 2PC protocol such as presumed abort or presumed commit [2, 69]. In this case, if no message
is received in a defined timeframe, then the actions are automatically either aborted or committed. However, the participant still loses autonomy until the timeout, and the atomicity of any multidatabase transaction is no longer ensured, as it is possible for some participants to automatically commit or abort some actions incorrectly because an error prevented a message from the transaction coordinator from being received.

To avoid these problems, the property of isolation is often not required for multidatabase transactions [20]. The local isolation property enforced for each of the individual systems involved in the transaction ensures that the behaviour of each participant is acceptable, and this also helps reduce the problems that the lack of global isolation can cause. However, a lack of global isolation can cause some transactions to fail when they should have succeeded. For example, if one transaction has booked the last seat on a flight and then another transaction also wants a seat on the flight, the second transaction will fail. If the first transaction later rolls back, releasing the seat, then the second transaction, which has already completed unsuccessfully, could have completed successfully, had it been given the now-unbooked seat. While avoiding such problems would be required for complete transactional support, the lack of global isolation allows each system to maintain more autonomy than full support would allow.

Thus having an occasional transaction fail when it should have succeeded is often seen as an acceptable alternative to an increased loss of autonomy.

2.2.3.2 Relaxation Of Atomicity

As well as isolation typically being ignored, atomicity is also often seen as too difficult to maintain. Each database in a multidatabase system may implement different concurrency control systems, or other different features that not all the databases have available, so that it is impossible to rely on any such extensions to help enforce atomicity. Instead, it is often suggested that atomicity should be replaced with something weaker, such as *semi-atomicity* [128] or, more commonly, *semantic atomicity* [50]. Semantic atomicity details that all databases should end up in a state in which an acceptable outcome has occurred, rather than requiring that such a state be maintained at all times. If a transaction succeeds, then the outcome is identical to that achieved by enforcement of atomicity. If a transaction fails, however, it is not necessary for the databases to return to the exact initial configuration. For example, suppose there is an
action that is compensatable, that is, it is possible to perform another action to undo the effects of the action in the transaction. In this circumstance, it may be acceptable to perform the original action so that any other transaction sees it as completed, but, if the transaction fails then the compensation action can be called to undo the effects, and all future transactions will not see the changes made by the failed transaction. If the compensating action only performs actions to make it appear that the original action never took place, then atomicity is preserved (though at the cost of isolation). However, there might be side-effects of the action; for example, if cancelling a flight, only a partial refund may be given as a compensating action. These side effects do not affect the consistency or durability of any of the databases, and still result in an acceptable state, but true atomicity is not achieved [72].

Younas, Eaglestone, et. al [125, 126, 124] claim that for multidatabase systems, particularly when distributed over the Internet, ACID properties of transactions should be replaced by the SACReD (Semantic Atomicity, Consistency, Resiliency, Durability) properties. The consistency and durability properties are identical to those provided by ACID transactions, while atomicity is replaced with semantic atomicity. Isolation is removed completely and the new concept of resiliency is added. Resiliency is the ability of a transaction to handle errors.

Some approaches to support resiliency include the provision of redoable actions that are guaranteed to eventually succeed if called a finite number of times. Another option that increases resiliency is to implement alternatives for parts of transactions. For example, if it is not possible to book a seat on a particular airline’s flight, it may be acceptable to book a seat on a flight offered by another airline. If using the first alternative fails, the next alternative(s) can be tried, until eventually either one of the alternatives succeeds, in which case the transaction can continue, or all alternatives fail, in which case the entire transaction is rolled back. The use of resiliency can greatly decrease the number of failed transactions in a system.

Resiliency is simply a further relaxation of atomicity. In the above example, the first airline would see the transaction as failing, while all other participants would see it as successful. Thus, while it does not appear as if the transaction never took place (as is necessary for an atomic transaction that fails), it also does not universally appear to succeed. As each of the systems are autonomous, this inconsistency of views should not
cause a problem. The first airline should be unaware of the other actions taking place in the transaction, and so should not be able to make assumptions based on the belief that those actions failed. In this way, resiliency helps improve performance of the overall system, without causing any real difficulties.

2.3 Summary

This chapter provides some necessary background information to understand transactions in a service-oriented architecture. An SOA is based on the use of simple services, descriptions of these services, and basic operations such as publishing, selecting, and binding to services. Web Services are introduced as the most common implementation of an SOA, and compared to previous technologies designed to provide similar functionality. The various Web Services standards can be used to support the basic operations of an SOA, as well as extra functionality to make the architecture more easily usable.

The chapter also investigated the concept of transactions as a way to support cooperation in systems where multiple parties work together. Transactions allow multiple actions, such as service calls, to be combined into a single well-defined executing unit. The ACID properties required of traditional transaction systems are discussed, as are advanced transaction models and relaxations that are often necessary to make transactions usable in multi-party systems.

The next chapter will build on this information to describe the current state of transactions in the Web Services environment. It will be argued that the Web Services environment can be considered as a multidatabase system and, as such, research from multidatabase systems can be directly applied. However it will also be shown that, because of some of the unique features of SOAs, traditional transactional methods are not always the best option, and enhancements only available in SOA environments will be discussed.
Chapter 3

Service Composition In The Web Environment

Competition has been shown to be useful up to a certain point and no further, but cooperation, which is the thing we must strive for today, begins where competition leaves off.

Franklin D. Roosevelt

The purpose of a service is to perform one specific task. However, a client often requires multiple services to cooperate in order to completely fulfil his or her requirements. One of the great hopes for Web Services is that such compositions will happen automatically, perhaps even without being explicitly requested by a user. For example, a client may wish to buy or sell some stock from a broker. Rather than contacting a specified broker, the client could call a directory Web Service that automatically searches for the broker offering the best price for the request. As the Web is a constantly changing resource, the set of available brokers would depend on which were active when the request was sent. Further, in some cases, a single service may not be available to perform the requested operation, but by combining multiple services the desired operation may be possible. Continuing with the trading example, imagine that the client wished to use a different currency than any of the brokers would accept. Rather than simply failing, it should be possible for the system to automatically find a currency conversion service, and use this in combination with the broker service to fulfil the client’s request.

As such an automated system does not yet exist, there is also a need to allow clients to manually combine different services into a particular workflow. A client’s needs may be
CHAPTER 3. SERVICE COMPOSITION IN THE WEB ENVIRONMENT

quite complex, requiring intricate logic to determine which outcomes are acceptable and which should be considered successful. A typical example is a holiday booking [101, 117]. The holiday requires a flight and some accommodation; neither should be booked if the other is unavailable. Depending on the distance of the accommodation from the traveller’s destination, the booking of a hire car or taxi may also be required. Even when the accommodation is close enough to the airport, the traveller may prefer to have a hire car if possible, but if none were available then the result of having the flight and accommodation booked should still be considered successful. In other words, the traveller’s workflow comprises two necessary components, and one component that is preferable, but optional when the accommodation is close to the airport.

This chapter first looks at the requirements of a system that allows such automatic and manual workflows to be created and processed. It is shown that one possible option for such a system is to support transactions. Thus, the current state of Web Services transactions is discussed in Section 3.2. Finally, this chapter introduces the higher-level concepts of orchestration and choreography in Section 3.3.

3.1 Requirements Of A Service Composition Platform

In many cases it is necessary for a client to utilise more than one service to achieve a required result. However, there are many different problems that can occur when trying to combine multiple unrelated services to work together for one specific task. Issues such as how the services are found (especially when the set of available services changes dynamically); how they can be combined to work for a specific aim; and how errors are handled, are all important when trying to compose a task from multiple services. Chakraborty and Joshi [31] identify five different issues that must be catered for by a comprehensive service composition platform in order to allow dynamic service compositions to be formed:

1. Service Discovery

The ability to find all services that provide the required functionality, regardless of how each is invoked. This requires the service discovery architecture to have semantic information about each service so it can determine which services are appropriate.

2. Service Coordination and Management
A method to efficiently and competently communicate with each of the services involved in the composition to ensure the correct result for the service composition.

3. Uniform Information Exchange Infrastructure

Different services may operate in different ways, but a service composition platform must abstract over these differences. The services involved in a composition may use different paradigms for communicating with clients, and it is up to the platform to abstract over these differences.

4. Fault Tolerance and Scalability

The system should handle faults properly, and gracefully degrade as more and more services fail or become unavailable. Further, as more users and services are added to the system, it should still continue to work efficiently.

5. Adaptiveness

Nothing on the Web can be assumed to be permanent. Thus a good platform will continuously use the services available at any time to provide the best experience possible at that time.

3.1.1 Service Discovery

Before multiple services can be combined into a service composition, it is necessary to find a service or services that can perform the required tasks. It is only after the necessary services are discovered that they can be combined to attempt to achieve the client’s needs. Service discovery essentially requires semantic information so the system can automatically determine which services (or combination of services) offer the functionality required by the client. The industry approach to Web Services compositions and the Semantic Web approach have been developed independently, meaning that much of the work on semantic Web Services is not available in the real world [109]. While UDDI [35] does provide some ability for automatic service discovery, industry-specific semantic information that goes beyond what is required by the UDDI and WSDL specifications is necessary for truly dynamic service discovery [49].

One way to provide the required semantic information is to include it with the description of each service. This information can be added using a formally-specified
language, making use of concepts such as ontologies developed for the Semantic Web [6]. Other approaches can use less specific textual descriptions, and then attempt to infer additional semantic information [61]. There are also techniques that remove the need for a central service discovery coordinator, allowing semantic information to be used to discover services in a peer-to-peer fashion [111, 12].

Truly dynamic service discovery is still an open problem for Web Services. The currently-defined standards do not require sufficient semantic information to allow fully automated service discovery, meaning that extensions or further specifications are required before services can be correctly categorised, and thus discovered. In many cases there is no clear way for a provider to explicitly list the semantic information of their service [49]. While there are techniques to automatically or semi-automatically discover semantics for Web Services [97], many of these require a detailed ontology for the specific industry in which the service exists, and most do not have this available.

3.1.2 Uniform Information Exchange Infrastructure

While Web Services standards do specify how Web Services are syntactically defined, it is still possible for providers to offer similar services in different ways. Services can take different parameters, have different names, or return information differently. However, Web Services also offer a workaround to allow a single approach to access these services. It is possible to define an abstract service [105] that knows how to interact with each of the different services that offer similar functionality. This abstract service, when requested by a client, contacts actual providers, converting the client’s request into the form that is understood by each particular provider. The abstract service then receives the response(s) and converts it/them into the format published by the abstract service before passing it back to the client. The client then simply uses the abstract service rather than any particular provider, and all providers known to the abstract service appear to the client to have an identical interface.

3.1.3 Adaptiveness

Given an adequate service discovery mechanism, abstract services could be generated automatically, and dynamically change as different services become available or cease to be usable. Further, the abstract service can offer additional benefits to the client by
employing different strategies to select which actual provider to use depending on the client’s requirements [102]. For example, the abstract service could always choose the cheapest provider, or the one that guarantees a particular quality of service, giving the client the best possible alternative from all that are available.

To allow true adaptiveness, it must also be possible to dynamically compose services when necessary to provide required functionality [81]. One approach to implementing dynamic compositions is to have semantic contracts with each provider [80]. Then, by utilising a good service discovery mechanism, an abstract service that performs its own service composition could be created to provide the functionality required by the client. Thus, the client would not need to add the complexity of fully detailing how a combination of services could be used to perform a task that it considers to be a single unit of work in its overall activity; the client would request the single service from the abstract service, and the abstract service would utilise as many services as necessary to provide that functionality to the client.

### 3.1.4 Other Requirements

A good service discovery mechanism, and the ability to dynamically create abstract services, are enough to cover most of the necessary issues required of a good service composition platform. The other issues are service coordination and management, and fault tolerance and scalability [31]. The previous chapter described how transactions allow multiple actions to be combined into a seemingly single action. Further, in advanced models, transactions can be arbitrarily nested, and each of these nested transactions can specify how it handles errors. Thus, it is possible to solve the remaining issues required for a service composition platform by supporting transactions in the Web Services environment. As will be seen in Section 3.2, the typical service coordination used with Web Services is separate from transaction support, with transactions offering extra functionality to handle errors. Since these coordination standards already exist, this thesis will not consider any other service coordination mechanisms.
CHAPTER 3. SERVICE COMPOSITION IN THE WEB ENVIRONMENT

3.2 Transactions In The Web Services Environment

Many standards have been defined to give some level of transactional support over the Web, for example [77, 103, 30, 92, 24]. Section 3.2.1 introduces the main standards, and shows how they have developed over time. These standards provide useful platforms for further development of transactions in the Web environment. Section 3.2.2 then introduces advanced transaction models that can be used with Web Services. While some of these move previous multidatabase transaction technologies to the Web, others, by taking advantage of the unique characteristics that the Web environment provides, offer completely new concepts.

3.2.1 Web Services Transactions Standards

Web Services allow multiple disparate systems to communicate with each other in a compatible way. Each service provider tracks its own state and is typically completely independent of all other providers. This makes the Web Services environment essentially a multidatabase system. Thus, all research from multidatabase systems may be used in the Web Services context with only minor modification. As with all multidatabase transactions, it is often too expensive (either in terms of reduction in level of service, or amount of work that has to be redone) to completely enforce all of the ACID properties, and it is usually atomicity and isolation that are relaxed [40]. The remainder of this section provides a brief introduction to the various standards for Web Services transactions, some of which support full ACID properties, and many of which relax one or more of the ACID properties.

3.2.1.1 Transaction Internet Protocol

One of the earliest standards for transactions specifically designed for use over the Internet was the Transaction Internet Protocol (TIP) [77]. TIP is a simple two-phase commit (2PC) protocol. Unlike earlier 2PC standards, TIP separates the transaction control protocol from the communication protocol used by particular applications [73]. This means that TIP can be used in the heterogeneous environment that the Internet provides rather than the homogeneous environment to which previous 2PC protocols had typically been restricted.
As TIP is a 2PC protocol, it fully supports the ACID properties. As a result, the protocol is excellent for short-lived transactions, however, the locking required by the protocol means that it cannot effectively be used for longer-running transactions [67]. Thus, while TIP cannot be used in all situations, it did introduce the idea of splitting the control of transactions from the rest of an application, and this important idea underlies most other transaction protocols for use over the Internet [95].

### 3.2.1.2 Tentative Hold Protocol

The Tentative Hold Protocol (THP) [103] is designed to support the automated coordination of transactions between multiple business entities. It is a loosely-coupled, message-based framework that allows businesses to negotiate and exchange information before a transaction actually occurs. THP is based on the idea of a non-blocking reservation. For example, THP can be used in a “shopping cart” for an on-line store. Numerous people can add an item to their baskets, but the item is not exclusively locked by them until they actually “check out”. When check out occurs, an enquiry is performed to ensure that the item is still available. If such a purchase prevents other clients who have a tentative hold on the item from being able to purchase it (i.e. because of insufficient stock), those other clients are sent a notification letting them know that their hold is no longer valid.

THP thus benefits both clients and service providers. Providers are able to offer multiple holds, and as a result clients are not required to wait for other clients to finalise their decisions before being able to proceed. Further, the provider is able to confirm a reservation with the first client who makes a final decision to take the requested resource, rather than the first to indicate that they may require the resource. As well as decreasing wait time, THP can also reduce the number of required cancellations, because a client can obtain tentative holds on all required resources before any resources are actually provided [46]. Since the provider notifies the client as soon as a tentative hold becomes invalid, it is less likely that resources become unavailable once all holds have been obtained.

### 3.2.1.3 Business Transaction Protocol

One of the earliest standards used specifically for Web Services transactions was the Organization for the Advancement of Structured Information Standards’ (OASIS)
Business Transaction Protocol (BTP) [30]. BTP does not actually require Web Services at all, and can be used in any similarly loosely-coupled environment. While this can be seen as an advantage in some circumstances, it is more commonly seen as a disadvantage for general Web Services transactions [76]. As BTP does not rely on any of the Web Services specifications, it had to respecify some details that were already defined by the Web Services standards. For example, all service dependencies must be defined in the specification because the Web Services infrastructure cannot be assumed. This both lengthens the BTP standard and makes it necessary for some features to be explicitly specified in two different places. Further, the business logic of a BTP transaction is included with the transaction protocol, so users have to be very closely tied to the coordinator of a transaction [23].

Despite these shortcomings, BTP does address the requirements for transactions in which atomicity and isolation are not strictly supported. There are two transaction types in BTP: atoms, and cohesions. Atoms are similar to ACID transactions (and in fact may be implemented as ACID transactions), but there is no way to determine the extent to which the ACID properties apply to an individual atom. The more interesting transaction type for Web Services is the cohesion. Many parties can join a cohesion; the coordinator of the transaction then queries the individual members to see if they can perform the required operations to complete the coordinator’s task. If enough members of the cohesion can successfully perform their action so that the result represents an acceptable end state, the coordinator informs them to do so. Any members of the cohesion unable to perform their task (or informed not to) may believe the transaction has failed, even though other members of the cohesion see the transaction as completing successfully. This means that a cohesion does not support atomicity; some actions succeed even though other actions in the transaction have failed.

3.2.1.4 Web Services Transaction (WS-TX)

The OASIS standard Web Services Transaction (WS-TX), comprised of Web Services Coordination (WS-Coordination) [93], WS-AtomicTransaction [91], and WS-BusinessActivity [92] (which deprecates WS-Transaction [38]) offer similar functionality to BTP, though these are built on top of Web Services standards. As such, the specifications are much shorter, and thus simpler, than those for BTP. Further, it
was realised that coordination can be useful in areas other than transactions, so Web Services Coordination can be used without Web Services Transactions, for example to provide a reliable multicast environment where all participants are guaranteed to be sent a message. Web Services Transactions build on Web Services Coordination to create similar transaction types to those defined in BTP. A Web Services transaction’s atomic transaction is similar to a BTP atom, though the atomic transaction is guaranteed to be atomic (with no specification of isolation). Having an atomicity guarantee improves interoperability between different transactional backends and is useful for short-duration interactions [76]. Web Services transactions also define Business Activities, which are similar to BTP cohesions [75].

While WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity have concepts that are similar to those in BTP, they make it much easier to separate business logic and the transaction infrastructure [76]. This is vitally important to making applications developed using these technologies more reusable. For example, BTP requires a 2PC protocol; there is no way that it cannot be used. By using WS-Coordination in WS-TX, however, it would be simple to replace 2PC with another protocol (three-phase commit [107], for example), without affecting the actual application. Thus, the WS-TX standards are more powerful than BTP, and are therefore more useful.

3.2.1.5 Web Services Composite Application Framework

After BTP and WS Coordination/WS Transactions were created, it was realised that there were still some abstractions that could take place, for which there was no provision in existing protocols [75]. The Web Services Composite Application Framework (WS-CAF) [24] is split into three parts, with each part utilising the more general part below it to perform its function. The first (lowest level) part is Web Services Context (WS-CTX) [25], which lets different Web Services share the same context with each other, each altering it as they see fit. The Web Services Coordination Framework (WS-CF) [26] places a coordinator in charge of the shared context, forcing Web Services to access the context through that controlling manager. Finally, Web Services Transaction Management (WS-TXM) [27] uses that coordinator to provide support for three different transaction types: TX-ACID, TX-LRA, and TX-BP. TX-ACID implements a standard ACID transaction. TX-LRA is used
for Long-Running Activities that cannot fully support the ACID properties because their runtime is too great. Finally, TX-BP is for Business Processes and is similar to BTP cohesions or WS-Transaction Business Activities [75].

Apart from the further level of abstraction, which allows WS-CTX and WS-CF to be used without WS-TXM, WS-CAF has little separating it from WS-Coordination and WS-Transactions (WS-CAF can be seen as a superset of those standards). Even the new type of transaction introduced in this standard, TX-LRA, can be modelled as a WS-Transactions Business Activity. In fact, it has been stated that much of the reasoning underpinning WS-CAF is actually political [75]. WS-CAF is still a useful step forward [28], especially with the work of the OASIS WS-CAF Technical Committee to make it an OASIS standard [89].

3.2.2 Advanced Transaction Models

The Web Services transactions standards do not always adequately meet the needs of users or providers of services. Numerous different expansions to these systems have been described or developed (e.g. [5, 14, 32, 46, 58, 74, 79, 98, 123, 127]). These extend Web Services transactions in some way, either trying to bring traditional transactional aspects to the Web Services environment, or looking at the differences that the Web environment offers and utilising them to the best possible advantage.

3.2.2.1 Transactions In The Web Services Environment

The Web Services transactions standards presented so far do not directly allow all of the advanced transactional techniques mentioned in the previous chapter. This is sometimes because the Web Services environment is not well suited to the techniques, but there are cases in which certain advanced models would be beneficial. Further, the Web Services environment is different from most multidatabase systems because it is loosely-coupled, and each provider is autonomous. This allows some methods that are not applicable in other multidatabase systems to be useful in the Web Services environment.

3.2.2.1.1 WS-Sagas And THROWS

WS-Sagas [14] extend the nested-sagas model [51]. Thus, each activity has a compensation activity allowing transactions to be aborted even when they have been
CHAPTER 3. SERVICE COMPOSITION IN THE WEB ENVIRONMENT

partially completed. Further, the model includes the concept of a vitality degree, which allows certain activities in the transaction to be categorised as non-vital. If a non-vital activity fails then a transaction as a whole can still succeed. If a vital activity fails, then an attempt is made to find a substitute Web Service to perform the same action. Only if no such alternative can be found does the transaction as a whole fail.

Transaction Hierarchy for Route Organization of Web Services (THROWS) [15] is an architecture for executing Web Services compositions arranged in WS-Sagas or similarly nested transactions. The two main components offered by THROWS are the Candidate Engines List, which keeps track of alternate services that can be used if a particular service fails, and the Current Execution Process concept, which tracks the progress of a composition and can delegate control of the process to particular engines in the Candidate Engines List.

3.2.2.1.2 Client Callback

The tentative hold protocol offers providers a way to allow more than one client to have a hold on a single resource at any one time. This can greatly increase usability of the system, as fewer clients have to fail immediately. However, when a client receives a tentative hold, it does not know whether or not the resource is exclusively its own. If the client completes a large amount of work before being informed that a hold has become invalid, the client may then need to expend a large effort to cancel the transaction. An alternative is for providers to indicate to a client that a resource is currently being used (either already allocated, or locked by another transaction), but later “call back” the client if the previous situation changes. Essentially, the client becomes a server that listens for a change of circumstances with the provider [98]. In this way, when the client receives information that a resource is unavailable, it is able to decide whether to wait to see if the resource becomes available, whether to find an acceptable alternative action, or whether to cancel the entire transaction.

3.2.2.1.3 Combining Transactional Properties

Transactions in the Web Services environment work by combining services from potentially multiple providers into a single action. Each of these providers may support different transactional properties, not all of which are necessarily compatible. Fauvet, et al. [46]
consider three different kinds of services:

1. Atomic: a service that provides an ability to reserve a resource, cancel a reservation, or commit a reservation

2. Quasi-atomic: a service that provides the ability to commit an action, and a compensation action to cancel a previously committed action

3. Non-atomic: a service that only provides the ability to commit an action, with no ability to undo that action

These services can be combined to offer differing levels of transactional guarantees. Firstly, operations can be grouped into sets, and it is possible to define a minimum and maximum number of operations that must succeed for the grouped operation to be successful. For example, if a transaction includes an essential action, that action will be grouped into a set on its own, and the minimum and maximum number of operations required for that grouping will be one. Such groupings will only ever be non-atomic if fewer than the minimum number of operations that must succeed are either quasi-atomic or atomic.

The above groupings are executed as follows: Initially, the actual services to be used are chosen at runtime based on which of those available provide the requested operations, and which is the “best” at that moment (with “best” being defined by a function specified when the transaction is created; typically the one with the lowest cost). Once all available alternatives have been discovered and ranked, each alternative is handled differently depending on the level of transaction support that it provides. If the service is atomic, the resource is reserved. If quasi-atomic, the action is committed. If the number of committed actions and the number of held resources is greater than the minimum number of operations required for the grouping to be successful, then all the atomic actions are committed and any non-atomic services are told to commit (with any of these that succeed being a bonus). If the number of committed actions is not greater than the minimum number required, all atomic services are cancelled and all quasi-atomic services are told to compensate.

3.2.2.1.4 Transactional Attitudes

Mikalsen et al. introduce a new way to ensure transactional reliability in the Web Services environment [79]. Their technique allows providers to indicate the “transactional attitude”
of the provider for each service it provides; that is, the kind of transactional support that
the provider offers for the service. Providers are able to offer multiple different attitudes,
three of which are defined in the Mikalsen paper [79]. These are *pending-commit*, *group-
*pending-commit*, and *commit-compensate*. Pending-commit allows a pending state for the
operation; a client can make their request, the provider can either refuse or say “yes”,
and, if the provider said “yes”, the client can later choose whether to commit or roll
back. The given example is a flight reservation, which can be requested and then later
confirmed. Group-pending-commit is similar to this, except that there may be more than
one operation in a pending state. An example of this is booking multiple legs for a taxi ride;
the overall booking is successful only if all required legs can be booked. Finally, commit-
compensate allows an already completed action to be later compensated. Providers are
also able to define different attitudes, which they can then utilise. The provider indicates
its attitude by extending the WSDL description of its services with WSTx transactional
semantics. For example, if an action can be completed and later compensated, the following
annotation would be added to its WSDL definition: 

\[
\text{<wstx:binding attribute="commit-compensate">}
\]

Once the WSDL has this additional information, clients are able to use the services
with the given transactional semantics. Further, clients can explicitly indicate their
transactional attitude by utilising a Web Service to create and manage a pre-defined
transactional pattern. Again, some patterns are included in [79], but clients are able to
create others if these prove inadequate. The pattern included in the paper is the *flexible
atom* attitude. This allows the client to indicate that some actions are critical and others
only optional.

After providers and clients have specified their transactional attitudes, the WSTx
middleware is used to automate execution of the transaction. The middleware must
support all of the transactional attitudes being used by the providers and clients, and
manages the context and transactional interaction between a client and a set of
providers. The middleware must also support persistent logs to keep track of all
transactions and their states; once the client hands the transaction over to the
middleware, it is up to the middleware to finalise the transaction even if errors occur.
CHAPTER 3. SERVICE COMPOSITION IN THE WEB ENVIRONMENT

3.2.2.2 Management Of Web Services Transactions

Since transactions in the Web Services environment are quite different from traditional transactions, it is necessary to develop new techniques to manage these transactions. There is typically no single coordinator for multiple transactions, and transactions can have long running times. In particular, failure in Web Services transactions is also quite different to failure in traditional transactions. Transactions can run for long periods of time, and multiple parties may either join or leave a transaction as it is running, so special care is needed to ensure that any failure is handled in a well-defined way.

While Web Services transactions typically ignore global isolation, in some cases, such as when independent transactions rely on each other (for example, two bank transactions altering the same account balance), enforcing the full set of ACID properties is still desirable. Some form of concurrency control is needed to allow fully ACID transactions to occur, and the traditional methods described in the previous chapter are not well-adapted for use in the Web Services environment [5]. Thus, various new schemes have been developed for this purpose (e.g. [5, 32, 58, 123, 127]). Further, even when complete ACID properties are not required, it can be beneficial to track the dependencies of currently-running transactions to ensure that certain errors do not occur [74]. Some techniques that offer this concurrency control and dependency tracking are introduced below.

3.2.2.2.1 The Web Services Transaction Model (WS-TM)

A model for Web Services transactions is proposed in [123]. This model is termed WS-TM and provides relaxed atomicity, allowing some participants in a transaction to see the transaction as succeeding while others see it as failing. These different views are predominately achieved through a relaxed-2PC protocol, which allows the successful completion of a transaction even when certain non-critical participants are unable to complete the action asked of them. The framework also supports compensatable actions and includes a simple context management system and transaction manager factory to initiate shared contexts for transactions.

Essentially, the WS-TM framework offers only some of the features available in standards such as WS-CAF, allowing consensus on the outcome of a transaction and the ability for some participants in a transaction to see the transaction as successful (and
thus commit their operations) while other participants see it as a failure (causing them to cancel) [123]. WS-TM is much simpler than the other standards, and is designed to be implemented in a compact environment. This has benefits when only the supported features are required, but the flexibility of previously-mentioned standards can mitigate some of the potential disadvantage that their complexity seems to imply. For example, with WS-CAF it is sufficient to have a provider support a single transaction type to render it usable by the coordinator. Thus the extra support and features offered by the other standards mostly outweigh the benefits of the simpler WS-TM framework.

3.2.2.2.2 Transaction Commit Protocol For Composite Web Services

Transaction Commit Protocol for Composite Web Services (TCP4WS) [127] is an attempt to bring the SACReD properties (Semantic Atomicity, Consistency, Resilience, Durability) to Web Services transactions. Essentially, TCP4WS has a central coordinator for the transaction; the coordinator communicates with subcoordinators for all subtransactions (with the simplest subtransaction being a single action). Each subcoordinator attempts to complete its transaction, and replies to the central coordinator with either a “commit” or “abort” message. If all subcoordinators return a commit message, the central coordinator commits the entire transaction. If any subcoordinator sends back an abort message, then the coordinator attempts to use an alternate subtransaction to perform the required task. If no such alternative exists, then the transaction must abort. To effect the abort, the central coordinator sends an abort message to all subcoordinators, which then perform the compensating actions for the actions they previously completed.

It is found that TCP4WS often outperforms Two-Phase Commit even when Presumed Abort is used [127]. This is especially the case when the transaction commits successfully without the need for any alternate subtransactions, or when message delay is high. The use of alternative subtransactions in the case of failure greatly decreases the performance of the protocol, but many more transactions complete successfully [127].

3.2.2.2.3 Jenova

Jenova [58] is described as a new approach to concurrency control for Web Services transactions. It is based on nested transactions and also includes admission control. An admission controller guards each Web Service to limit the number of concurrent
transactions, to a capacity that the controller can handle. This reduces the errors that occur simply because a Web Service is too busy to respond to any new requests, which, it is claimed, greatly reduces the cost and number of roll backs in the system.

Jenova separates transactions into two kinds: primitive, a transaction that interfaces with only one Web Service; and complex, a transaction that interfaces with two or more Web Services. Each complex transaction is made up of subtransactions (either complex or primitive), thus allowing nested transactions. Jenova assumes that the underlying Web Services concurrency control mechanisms are serialisable [56]; the outcome of the execution of a set of concurrent transactions is equivalent to some serial execution of the transactions, with, at any time, only one transaction in progress. Each primitive transaction is defined by a set of preconditions as well as an action to be performed. Before performing a primitive action, the admission controller must first be contacted to obtain permission to call the Web Service. The admission controller checks that the Web Service call will not violate any conditions that must be maintained to ensure the currently running transactions can succeed. If the preconditions of the currently running transactions mean that the new transaction would be incompatible, the admission controller denies access to the Web Service until any such conflicts are resolved. This stops the client transaction from being able to continue, ensuring that the preconditions are not violated.

According to the Jenova concurrency mechanism, each Web Service transaction is run in two phases: simulation and commit. During the simulation phases, the actions of the transaction are performed, with a log file being used to record entries. When the commit phase begins, the actions in the log file are loaded and durably executed. The admission controller is used in the simulation phase. Since no work is actually done until after this phase is complete, the cost of rolling back a transaction is greatly decreased. It is only in the commit phase that actions are actually performed. This is similar to any two-phase commit protocol, so Jenova has the same problems that any such system has: individual service providers lose some of their autonomous nature to ensure that they are only accessed through a Jenova controller that complies with the protocol [76].

It was found that Jenova works well provided that the actual work done by the transactions hold resources for only a short period of time [58]. This is the case when most of the transaction’s time is spent waiting for an event (such as user input) to occur. It was found that in this situation, use of Jenova greatly decreased both the rate of
failure and the average cost for rolling back a transaction. These results seem consistent with what should be expected of two-phase commit protocols, though how Jenova performs when a transaction holds resources for a longer period of time was not tested. If all the actions in the transaction complete quickly then the transaction does not tie up resources for very long, so other transactions can use them more quickly. However, if just one action ties up resources for a longer period of time, then the resources guaranteed to be available to the other actions in the transaction cannot be used until either the transaction commits (in which case it uses the resources) or rolls back (in which case others are then able to use the locked resources). Thus, since Web Services transactions are often long-running [95], Jenova does not offer a complete practical solution for Web Services transactions.

3.2.2.2.4 Web Services Transaction Dependency Management Protocol

A scheme is presented by Choi, et al. [32] which composes subtransactions to create a global transaction, where each subtransaction is a Web Service located at some site. Thus, only subtransactions actually modify any database objects. For this model, the notion of dependencies is important; a completion dependency is defined whenever a subtransaction of one global transaction reads a value updated by a subtransaction of a different global transaction. In such a case, the transaction that caused the completion dependency is only committed if the other global transaction is committed. This situation is represented as a graph in which the nodes are transactions and the edges are completion dependencies from the dependant transaction to the dominating transaction. When a transaction commits, it is removed from the graph, but a transaction can only commit if it does not have any outgoing edges in the graph (that is, it has no completion dependencies).

It is thus possible for transactions to cause a cycle of completion dependencies, defined as a universal transaction. These universal transactions cause a deadlock if every transaction in the cycle is required to wait for all the transactions they depend on to finish before committing. This is overcome by a service designed to detect such circular dependencies; the service forces some transactions to roll back until the cycle of dependencies is broken. Even with this deadlock detection, however, it may still be necessary for a transaction to wait for another transaction to complete, before the waiting transaction can commit. If the executing transaction is long-running, then this
can cause a significant imposition for the waiting transaction. Further, any resources that are being held by the waiting transaction will remain locked until the waiting transaction can continue. Thus any new transaction that requires resources held by the waiting transaction will also have to wait for the long-running transaction to complete, causing more and more transactions to be reliant on the completion of the original transaction.

This scheme, which is further specified in [33, 68], is actually a more specific version of DCC [10, 63]. However, whereas DCC can be used to prevent both dirty reads and lost updates, the method described by Choi, et al. can only prevent dirty reads, because the method does not track concurrent writes when multiple service calls access the same data. As a consequence, it is still possible for errors to occur if two or more transactions modify the same value. While it would be possible to extend Choi’s method to track updates as well as data reads, that is not the only problem with the model; information about running transactions is shared [5]. This information could be private, but it is necessary for the different service provider’s transaction managers to know the information to create their dependency graphs. It would thus be possible for somebody to log the information a transaction manager uses to discover when and how particular clients are using other services. In this way, a competitor could unfairly discover inside information about their competition’s internal behaviour and perhaps even information about their underlying system.

3.2.2.2.5 Alrifai And Dolog’s Transaction Dependencies Management

Alrifai and Dolog’s work [5] attempts to support concurrency control while avoiding the problems mentioned in section 3.2.2.2.4. Their technique implements concurrency control running on the service provider side, without any need to share information about running transactions with other service providers. The system is based on the Distributed Serialization (Conflict) Graph Testing Protocol, using a graph similar to that used by Choi, et al. [32]. The difference is that, in this system, service providers are not aware of any transactions’ use of services offered by other providers.

Whenever a transaction coordinator specifies that a transaction should be committed, the service provider first checks if the transaction has any completion dependencies by determining whether it has any outgoing edges in the dependency graph. If the transaction
has a completion dependency for an unfinished transaction, the service provider sends a
*wait* message to the coordinator, and blocks until the dependency is removed. If, on the
other hand, there are no such edges, the actions on the service provider are performed
and a *completed* message is sent to the coordinator. The coordinator can then send back
a *close* or a *compensate* message, in which case the service provider either commits or
compensates the job, respectively, and then removes the transaction from the graph (thus
allowing any transactions depending on the newly completed or aborted transaction to
resume).

The example used by Alrifai and Dolog to show the kinds of problems their system
avoids involves transferring money from accounts. Assume that there are three
transactions, \( T_1, T_2, \) and \( T_3 \). \( T_1 \) credits account \( A \) with $300, then debits $300 from
account \( B \). \( T_2 \) debits $200 from account \( A \) and then credits account \( C \) with $200. \( T_3 \)
debits $100 from account \( C \) and then credits account \( B \) with $100. It may seem strange
that \( T_1 \) performs the transfer in the opposite way to the other two transactions (that is,
it deposits first whereas the others debit first), but it must be remembered that these
transactions may be run by completely different entities, which may indeed do things in
a different order.

For example, assume that all accounts begin with $0, and \( T_1 \)'s first action (depositing
$300 into account \( A \)) has completed. If \( T_2 \) then transfers money from account \( A \) to
account \( C \), followed by \( T_3 \) transferring money from account \( C \) to account \( B \), then all
accounts would have a balance of $100, and \( T_2 \) and \( T_3 \) will have completed successfully.
\( T_1 \)'s final action should then fail, since account \( B \) does not have the required $300. Thus,
\( T_1 \) should be undone, but as the money has already been sent to different accounts, that
is impossible. Using the described transaction dependencies management technique, the
fact that the first actions of \( T_2 \) and \( T_3 \) relied on a resource already changed by \( T_1 \) would be
detected before the actions of \( T_2 \) and \( T_3 \) were performed. Transactions \( T_2 \) and \( T_3 \) would
be prevented from continuing until \( T_1 \) had finished. As \( T_1 \) cannot be successful if all initial
accounts are $0, none of the transactions would complete and the $300 would not be lost
from the first transaction.

The main problem with Alrifai's technique is that it requires a detailed knowledge
of conflicts between actions. This knowledge is required to determine whether or not a
transaction has a completion dependency on another transaction. With this knowledge, it
is possible for individual service providers to calculate any such dependency by tracking which services have been invoked on the resources that the provider is offering [4]. The amount of information that the provider must store to track the completion dependencies is thus proportional to both the number of services offered by the provider and the number of resources that have been accessed, which could be quite large. The other problem, inherent in all systems that correctly support isolation, is that some transactions may be required to wait for other transactions to succeed before they can themselves complete. If just one of the transactions is long-running, then it may be necessary for all other transactions to wait for that long-running transaction to finish before they can complete. One attempt to solve this problem is to implement a non-blocking scheduling mechanism before invoking any services [3]. By having this initial step calculate estimated or guaranteed processing times, both clients and providers are better able to plan their transactions. This requires the processing time for an operation to be known before processing begins, which is not always possible.

### 3.3 Orchestration And Choreography

Consideration of the Web Services Architecture Stack in Table 3.1, shows that the transaction layer builds on many previous layers. SOAP [59] provides the general formatting and definition of how messages are sent. WSDL [34] builds on this to give a description of the basic interface of a Web Service. The registry (which could possibly be UDDI [35]) then allows the publishing and discovery of these WSDL descriptions. The security layer (possibly WS-Security [71]) allows authentication of participants and provides a guarantee that messages have not been modified or forged, and the reliable messaging layer (which is often provided by WS-ReliableMessaging [90]) ensures that messages are delivered between the participants. Transactions can utilise all of these layers to provide the required transactional guarantees. There are also two layers that build on the transaction layer: the business process languages layer, which describes the definition of orchestrations, and the choreography layer [122], which describes choreographies.

Orchestration and choreography are higher levels of service composition. A service orchestration is a local point of view of multiple service composition for a single
PARTICIPANT IN THAT COMPOSITION. A CHOREOGRAPHY, ON THE OTHER HAND, LOOKS AT THE GLOBAL
PICTURE OF THE INTERACTIONS AMONG ALL OF THE PARTICIPANTS IN THE COMPOSITION, WITH NO
CENTRAL POINT OF CONTROL. WHILE WEB SERVICES TRANSACTIONS STANDARDS CAN OFFER
ORCHESTRATION OR CHOREOGRAPHY CAPABILITIES, ORCHESTRATION AND CHOREOGRAPHY CAN EXIST
WITHOUT TRANSACTIONAL SUPPORT. THUS THESE CONCEPTS CAN BE INDEPENDENT OF TRANSACTIONS
AND ARE THEREFORE CONSIDERED SEPARATELY FROM TRANSACTION CONTROL TECHNIQUES.

3.3.1 Orchestration

An orchestration provides a precise way in which messages will be exchanged between the
participant executing the orchestration and the other participants with which it interacts.
Thus, an orchestration provides a single participant with explicit instructions on how to
achieve its goal. The following section describes the Web Services standard for describing
orchestrations.

3.3.1.1 Web Services Business Process Execution Language

The OASIS standard for describing Web Services orchestrations is the Web Services
Business Process Execution Language (WS-BPEL, or BPEL) [88]. BPEL is used to
formally describe business processes and business interaction protocols. BPEL also
includes the declaration of relationships to external partners, how to handle various
conditions, and how data is processed. Process data provides the state of a business
process, comprising typed variables that are either private to the process or come from a
message exchange with a partner. Partners can be considered as communication links
with an external party. In particular, there can exist a set of partner links, all for the
same partner.

Once the above are defined, the main actions of the business process are defined as

<table>
<thead>
<tr>
<th>SOAP</th>
<th>WSDL</th>
<th>Registry</th>
<th>Security Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reliable Messaging Layer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Context, Coordination, and Transaction Layer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Business Process Languages Layer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Choreography Layer</td>
</tr>
</tbody>
</table>

Table 3.1: The Web Services Architecture Stack [122]
a BPEL activity. These can either be basic activities, that simply receive a message or manipulate some data, or structured activities that define business logic between other activities contained in the structured activity. One of the most basic of the structured activities is the *sequence*, which executes a collection of activities in lexical order. It is also possible to conditionally run activities using the *if-else* activity, or repeatably run activities using the *while*, *repeatUntil*, or *forEach* activities. Activities that allow parallel processing exhibit slightly increased complexity. The *flow* activity executes a collection of activities in parallel; there is a variant of the *forEach* activity that works in parallel; and *event handlers* are run in parallel when certain conditions are met. Parallel processes can later be brought together using *join conditions*.

Exceptional circumstances may often prevent these business activities from completing successfully, for example requesting a funds transfer from a bank service may result in an insufficient funds fault. BPEL caters for this by providing *fault handlers*. These fault handlers are used to *catch* any fault situation that occurs, and then implement an activity to handle that fault. In addition to fault handlers that are invoked when a particular process fails, it is also possible to have *compensation handlers* for use when a partially-completed activity incorporates a later required activity that fails. These compensation handlers contain logic to undo the already completed parts when such a later fault occurs.

### 3.3.2 Choreography

While an orchestration describes an explicitly executable process for a particular participant in an interaction, a choreography provides a global overview of all participants. Rather than being directly executable, a choreography defines how the various participants in the interaction can communicate with each other. Thus, given an orchestration for each participant, it is possible to ensure that the described interactions follow the rules of the choreography, but it is not always possible to extract the desired orchestrations from a choreography description. Whereas WS-BPEL is the main standard for defining Web Services orchestrations, the Web Services Choreography Description Language is the proposed recommendation for describing Web Services choreographies [122].
3.3.2.1 Web Services Choreography Description Language

The Web Services Choreography Description Language (WS-CDL) [122] allows the description of a global multi-party contract for a particular composition of services. It defines a formal language for the expression of rules for communication between the parties. Often, such rules are expressed in a native language, such as English. By allowing rule expression to be more structured, WS-CDL makes it possible to better verify various compositions and their properties.

In WS-CDL, information is exchanged between participants (possibly across trust boundaries) who are acting in particular roles. Participants are thus linked through relationships that indicate which role(s) they should each take. Participants can then send the required information over communication channels based on other participants’ roles. Such interactions can then be ordered and structured in activities, which can be run sequentially, in parallel, or conditionally. activities can also be nested, allowing very complex interactions to be defined.

These choreographies are placed in packages, which can then be reused for multiple different interactions, provided, of course, that the interactions behave in a similar way. It is possible to specify different participants to take on different roles, but use an existing choreography to specify well-defined behaviour of the composition, thus ensuring that the final result is acceptable (i.e. that all interactions are both allowed and reasonable).

3.4 Summary

This chapter presented a set of ideals that should be provided by all comprehensive service composition platforms. The first of these is the ability to discover services that can meet a required need. This problem has been the subject of previous research, but truly dynamic service discovery is still an open problem for the Web Services environment. The other requirements of a good service composition platform for the Web Services environment can be handled by the concept of an abstract service, and by Web Services transactions. The abstract service can provide a consumer with an identical interface for all similar services and, by monitoring the current state of the system, can adapt to system changes. Web Services transactions then allow the various interactions to be coordinated and managed, and allow the correct handling of errors when they occur.
There are many standards for Web Services transactions because, over time, various new standards and improvements have been developed. The simplest techniques move previous non-Web research to the Web environment. Thus, there are standards that allow traditional 2PC transactions. More complex standards that support other previously-known service composition techniques have also been developed, and some novel ideas have been designed specifically for the Web and similar environments.

Sitting above the notion of transactions are the concepts of service orchestration and choreography. A choreography provides a global overview of message flow between all participants in the system, and an orchestration provides an executable plan for a single participant, describing how messages are exchanged between it and the participants with which it communicates.

The remainder of this thesis will concentrate on the transaction level, though other levels will be discussed as required. The next chapter examines the possible levels of transaction support that can be offered by a provider, distilling each level to the basic operations that are required for that support. The chapter will then describe a mechanism to allow providers to alter the level of transaction support to offer for a particular service each time the service is called. It will also specify how a client can combine operations with different levels of transaction support to create a transactional workflow that is guaranteed to satisfy the client’s needs.
Chapter 4

Dynamic Contracts For Web Services Transactions

If the administration wants cooperation, it will have to begin to move in our direction.

Mitch McConnell

Web Services transactions allow clients to combine services (possibly offered by multiple providers) into workflows that have well-defined behaviour even when some of the included service calls fail. However, the systems described in Chapter 3 give service providers very little choice in the level of transaction support that the provider can offer; in the best case a provider can specify the level of support offered for a particular service only when the service is defined. While this does allow more support for cooperation than the naturally-arising accidental cooperation built in to service-oriented architectures, the lack of freedom afforded by these systems essentially forces the provider to offer a particular level of cooperation, which it may not always wish to support. This chapter provides a high-level introduction, which will be more formally described in Chapter 5, of an enhancement to current systems that allows a provider to choose the level of transaction support it wishes to offer each time a service is called. This enhancement facilitates deliberate cooperation between service providers and clients by allowing each provider to dynamically alter the level of transaction support it offers for a particular service call, depending on the circumstances of both the provider and the client.

There are numerous standards already available for Web Services transactions (e.g.
These allow clients to combine services offered by various providers, possibly with different services offering different levels of transactional support. For example, it is possible to combine an Atomic Transaction [91] with a Business Activity [92] to give the client an overall transaction that at least provides semantic atomicity [50], but can also offer stronger atomicity for the providers that support the stronger level.

These main standards, however, do not allow many different kinds of transactional approaches to be defined and combined [57]. One attempt to overcome these problems is Mikalsen’s Transactional Attitudes [79]. This allows both clients and providers to specify the level of transactional support they require or offer, allowing the client to combine whatever level of transactional support is being offered into a transaction that provides a guaranteed outcome with which the client is satisfied.

Upon considering this work, we have identified some general transactional interactions that can be offered by a service provider. These are described in Section 4.1. We suggest that, when a client wishes to create a transaction, the system should first find a set of providers and determine the level of transaction support that each will offer for the required services. The client can then attempt to combine these services into a transaction that guarantees an acceptable outcome; what is deemed acceptable will depend on what the client is attempting to achieve and the level of risk it is willing to take. To allow the client to combine services with different levels of transactional support, a transaction coordinator is required. The remainder of this chapter will assume that the client is the transaction coordinator, and its transactional behaviour is described in Section 4.2.

The main limitation of Mikalsen’s Transactional Attitudes is that the level of transaction support offered for any particular service can only be specified once, in the definition of the service, and cannot adjust as the provider’s circumstances change. Consider, for example, a provider that has plentiful resources; say a hotel with 100 rooms available. If a client comes along and requests a single room, the provider may be willing to provide a stronger level of support (perhaps semantic atomicity) than if the client were requesting all 100 rooms (where the provider may prefer to offer a tentative hold). If the client only requests a small number of resources then the provider does not risk much in offering a higher level of transaction support; the likelihood of another client coming along and wanting the same resources is fairly low. If, however, the provider were to offer semantic atomicity for the larger request, then any subsequent requests for any
number of rooms would fail until the original client’s transaction had completed. If the original client ends up ordering all of the rooms, then that is the correct action from the hotel’s viewpoint. If, on the other hand, that client eventually cancels the booking, then the hotel still has all its rooms available even though other clients may have requested some of them during the time that the original client was determining whether or not to go ahead with their booking. Using transactional attitudes, it is only possible for the provider to offer one of these levels. Thus, it would be better if a provider could use any information at its disposal, such as the number of rooms it currently has locked, the previous behaviour of a given client, or the price a client is willing to pay, to determine which level of transaction support the provider would like to offer for a particular request.

As described above, providers have all the power when determining the level of transactional support they wish to offer. However, each provider offers their services because they want those services to be used. Thus, in some cases, if a slight change of transactional support can entice a client to use the services of that provider rather than of a competitor, it may be worthwhile for the provider to accommodate such a change. This requires a negotiation phase between a client and the service providers it wishes to use in a transaction; such a phase is described in Section 4.3.

Of course, many existing services do not currently offer the level of support required to allow the dynamic negotiation of transactional support required in the above-mentioned system. It is important that these legacy services are still able to be utilised by clients wishing to use dynamic transactions. Section 4.4 describes how this objective can be achieved. Finally, Section 4.5 compares the dynamic transaction scheme introduced in this chapter to existing practises in service transaction management.

4.1 Provider Contracts

When providers offer a service, there is a level of transactional support that they offer along with that service. In the simplest case, there is no explicit transactional support offered; the service either completes successfully or fails. However, more advanced cases allow the provider to deliver information or guarantees to a client before the client utilises the service, or allow the client to perform the operation but later cancel it if required.
This section discusses the levels of transactional support a provider can offer.

4.1.1 Possible Interactions

This section describes some general interaction patterns that can be offered by a provider to give varying levels of transactional support. These are based on the five basic operations of commit, enquire, prepare, compensate, and callback [100]. Of these, only commit is required; it actually performs the operation. Enquire checks whether the provider could currently fulfil a request, but gives no guarantee that a subsequent request will succeed. Prepare is similar to enquire, but guarantees that a subsequent request from the client (received within a timeout period) will succeed. Compensate undoes a previously committed operation, possibly at a cost. Callback occurs if either the provider is currently unable to support the request but may later contact the client if the situation changes, or if a previously successful enquiry may now fail.

4.1.1.1 Commit

The simplest case has the provider only offering the ability to commit the requested operation. Such a request can either succeed, in which case the operation is completed and it cannot be undone, or fail, in which case the operation is not completed. For example, if the provider is a hotel, the commit action would be the booking of a room.

4.1.1.2 Enquire/Commit

Placing an enquiry stage before the actual commit gives the client an idea of whether or not a request is likely to succeed. Rather than directly sending a request to commit, the client instead first asks whether their request would succeed. For example, a client may first ask a hotel whether a room is currently available before actually making a booking. If the client receives a negative response, then the client knows that the action is likely to fail if requested to commit, so the client can either find an alternate provider, or take another action to avoid the failure. If, on the other hand, the provider returns a positive result to the client, then the client knows that the request would have succeeded at that particular point of time, and thus has more confidence that the action will succeed if ever the client actually requests for it to be committed. However the provider gives no guarantee that any later request will succeed, just that the provider would currently be willing to provide
the requested service. The previously-mentioned Tentative Hold Protocol [103] can be described as an Enquire/Commit scheme.

4.1.1.3 Prepare/Commit

The presence of a prepare stage before the commit results in a traditional transactional pattern. If the initial prepare stage fails, then the client knows that the action will not succeed. However, after a successful prepare, the client is ensured that a call to commit will succeed. After the prepare call, the client can choose to cancel before calling to commit, in which case the provider can unlock its resources and again make them available for other clients. Thus, the client has no risk that the action will fail, but the provider has the extra cost of ensuring that such a failure is impossible. In this case, if a client prepares a room booking at a hotel, the provider must wait until the client has determined whether or not the booking should go ahead, and is only able to offer the room to other clients if the client decides to cancel the booking.

While in a traditional system such a guarantee would have no time restrictions, it makes sense for the provider to include a time limit in a service-oriented environment, since transactions typically run for longer periods of time, and the various providers would prefer to maintain some autonomy. Once this time limit expires, the provider can assume either that the client wishes to commit the action, or that it should be cancelled, depending on how the contract was initially decided.

4.1.1.4 Enquire/Prepare/Commit

Placing an enquiry before a prepare/commit pattern achieves the advantages of both the enquire/commit and prepare/commit patterns, and the provider must take into account the considerations for both of those patterns. From the client’s point of view, the enquiry stage is only worthwhile if there is some extra cost associated with the prepare/commit pattern without the enquire step; there could, for example, be a time limit on how long the prepare stage can be held without a commit. In this case, it may be best for the client to use an enquiry until it is truly ready to continue. Otherwise, the best option for the client is to simply use the prepare/commit pattern.
4.1.1.5 **Commit/Compensate**

Allowing the client to compensate an already committed action leads to provision of semantic atomicity. The client makes the initial request for the transaction to commit. If this fails then the action has failed. Otherwise, the action has succeeded. If the client later changes its mind, however, it can then ask to compensate the action, making it appear as if the action never succeeded, or at least attempting to do so. This is similar to the prepare/commit pattern, but the results of the call are immediately viewable by other parties; a call from another client to use resources held by the original client would immediately fail, whereas, when a prepare/commit pattern is used, the provider may instead wait to determine whether the original client actually used the resources before replying. Thus, a hotel would immediately inform any other client requesting a room that had been booked using a commit/compensate scheme that the room was unavailable. The hotel would only indicate that the room was available to new clients after a compensate action had occurred.

4.1.1.6 **Enquire/Commit/Compensate**

Placing an enquiry before the commit of a commit/compensate pattern results in a pattern with the features of the enquire/commit and the commit/compensate patterns. Similarly to the enquire/prepare/commit pattern, the initial enquiry is only useful if there is a time limit or cost associated with the compensate action.

4.1.1.7 **Prepare/Commit/Compensate**

Placing a compensate activity after the commit stage of a prepare/commit pattern indicates that there is a time limit associated with the prepare stage. If this were not the case then the compensate stage would never be necessary, as the client would only commit the action if it was definitely required. This means that once the time limit on the prepare stage has expired, the provider will offer semantic atomicity using the commit/compensate pattern.

4.1.1.8 **Enquire/Prepare/Commit/Compensate**

Placing an enquiry before the prepare of a prepare/commit/compensate pattern results in a pattern with the features of the enquire/prepare/commit and the
CHAPTER 4. DYNAMIC CONTRACTS FOR WEB SERVICES TRANSACTIONS

prepare/commit/compensate patterns.

4.1.1.9 Adding Callbacks

After each of the *enquire*, *prepare*, and *commit* steps of the above patterns, the provider can offer a *callback*. This means that, in the case of a failed response, the provider may later contact the client if a successful response could now be offered. For example, if a client was unable to book a room from a hotel that was fully booked, the hotel may notify the client if ever a room becomes available. A callback could also be sent after a successful enquiry to let the client know that the situation has changed; such a callback removes the need for the client to continually poll the provider to see if their query is still successful. When the client receives a callback after a failed action, it can choose to either accept, deny, or ignore the callback request. If the client accepts the request, the provider and client can continue their interaction as if the failed response had never been sent, instead being replaced with a successful response. When a client denies a callback request their interaction with the provider for the requested service terminates as if the callback had never occurred. If the client does not respond within a given time period, the provider will behave as if the request had been denied.

4.1.1.9.1 Callback After The Enquiry Stage

If the provider had initially informed the client that it would not have been able to complete the client’s request at the time the enquiry message was sent, a callback indicates that the client’s enquiry would now succeed. If, on the other hand, the initial enquiry was successful, a callback indicates to the client that the situation is changing. This may mean that the resources are no longer available, but may also be used by the provider to give preferred clients a warning that another client is attempting to use those resources; the provider could give the preferred client a short time to decide whether to continue with the transaction before replying to the other client. In either case, the provider gives no guarantee that a future call from the client will give the same result; all the callback indicates is that the previous situation has changed.
4.1.1.9.2 Callback After The Prepare Stage

When a provider offers a callback after a client’s call to prepare has been denied, the provider must guarantee that the client’s request to prepare will be successful for at least a short period of time. This may require the provider to lock resources for the client as if the prepare stage had already been successfully completed. If the client accepts the callback, then the transaction continues with the prepare stage having been successfully completed. The provider stops guaranteeing the resources for the client only if the client denies the callback request, or if the client does not respond to the callback before a timeout occurs.

4.1.1.9.3 Callback After The Commit Stage

Similarly to a callback after a prepare stage, a callback after a failed commit must ensure that, if the client accepts the callback, the given request can be committed. Unlike a callback after the prepare stage, however, the action may occur immediately and be viewable by other participants in the system. In this case, if the client does not accept the callback, the action must be undone. Otherwise, the action is performed immediately. In particular, if there is no compensation stage, a client’s acceptance of such a callback irrevocably performs the client’s requested action.

4.1.1.10 Resilience

As introduced in Chapter 2, resilience [125] allows a client to retry a failed service, possibly through a different service provider, in the hope that the new attempt will succeed, and thus avoid an error. The concept of resilience cannot be described using the basic operations introduced above. However, replacing concrete services with abstract services [105] can allow resilience in a way that is transparent to both clients and service providers. Clients use the abstract services rather than the services offered by the concrete providers, and the abstract service acts as a broker between all the providers offering alternative services, only notifying the client of a failure if all alternatives fail. These abstract services can thus be used in transactions described by the operations defined above. For example, an abstract hotel booking service may attempt to book a room at a number of hotels in a given city, and only fail if none of those hotels had any rooms available.
CHAPTER 4. DYNAMIC CONTRACTS FOR WEB SERVICES TRANSACTIONS

4.1.2 Offering A Transactional Contract

When a provider receives a request from a client, it must decide on the level of transactional support it wishes to provide. This is not always an easy decision, as many factors can affect the level of transactional support that a provider is willing to offer. Section 4.1.2.1 describes some of these factors, and how they can be used to help a provider choose the level of transactional support to offer for a given request. Once a provider has decided on the level of transaction support it is willing to provide, it must be able to notify the client of the decision. The client should then be able to negotiate with the provider to determine if a level can be found that is acceptable to both sides. Section 4.1.2.2 defines how this can be handled for this transactional scheme.

4.1.2.1 Choosing A Level Of Transactional Support

Offering transactional support has a cost to a service provider. Having to guarantee certain conditions may slow other interactions, or even stop them from being successful, and may require more processing than when no transactional support is offered. However, a higher level of transaction support may also be beneficial for providers. Clients may prefer working with providers that offer better transactional guarantees, and may even include optional components to their workflow only if the appropriate level of transactional support is available. Offering better transactional support may attract more clients to a provider, or keep existing clients happier than if weaker transaction support was offered.

Thus, there is a balancing act for a provider to determine the level of transactional support it wishes to offer for a particular request, and many different factors can influence the outcome. For example, the resources a provider has available, the current server load, the time of day, the nature of a client’s request, and the historical behaviour of a client may all influence the provider’s decision. This section explores some of these factors, and how they may affect the level of transaction support offered by a provider.

4.1.2.1.1 Default Transaction Level

A provider must first decide on the default level of transactional support it wishes to offer for a service. This establishes a baseline, and the provider’s state at any particular time can influence whether a stronger or weaker level of support will be offered. The default level somewhat depends on the service being provided; some actions cannot be undone,
and so no compensation action is possible, for example. However, the bulk of the decision is left up to the providers. They are free to offer whatever level of transaction support they are willing to support, with what is best typically being decided by what would draw in the most clients for the highest profit.

4.1.2.1.2 Current Resources

One of the factors that would influence a provider’s choice of level of transactional support is the current resources (or product) available to the provider. A provider with a large number of resources (such as a hotel with many rooms available) may be willing to offer a higher level of transactional support than one with a lower number. This is because a single client holding resources that are eventually released is unlikely to inflict a significant negative impact on the provider with more resources, but locking resources when only a small number are available may cause later clients to incorrectly fail and be much more catastrophic for a provider. The transactional guarantees that have already been given are also important. If a large number of clients have been granted a tentative hold on some resources, for example, then offering a higher level of transaction support, such as semantic atomicity, to a later client would cancel all of those tentative holds, which may be undesirable.

The nature of the resources being offered also affects the level of transaction support that a provider may choose to offer. If the resources are cheap and plentiful, the provider would probably prefer to offer a lower level of transactional support than for resources that are rare or expensive. As clients would be less willing to risk having a call to an expensive service fail than to a cheaper service, the provider would be more likely to tempt clients to use the expensive service with a higher transactional guarantee than the cheaper service.

4.1.2.1.3 Usage History

Information about how the service has been used in the past can also be used to help a provider choose the appropriate level of transactional support. If many requests come during a certain time of the day, or month, for example, then the provider may offer weaker transactional guarantees during that time in an attempt to have more clients successfully utilise the service. For example, a hotel may find that more rooms are booked for weekends than during the week, and thus offer a lower level of transactional support for weekend
bookings because it is likely that future clients would use the resources if the current client eventually decides against doing so.

4.1.2.1.4 Client History

A particular client’s historical use of a service can also affect the level of transaction support a provider is willing to offer. To help encourage loyalty, a provider may offer a higher level of transactional support to a repeat customer (such as customers in a hotel’s VIP program) than to a new client. Conversely, the provider may attempt to attract new clients by offering them a higher level of support ("special deals" for new customers). Further, a provider may offer a lower level of transactional support to a client who has previously cancelled many of their interactions with the provider, or may offer a higher level to those who frequently follow through and successfully complete interactions with the provider.

4.1.2.1.5 Other Factors

This section has introduced some of the factors that may influence the level of transactional support a provider is willing to offer, but there are many more possibilities. For example, information about competing services, and the level of transactional support they offer, could help alter a provider’s decision. The main point is that, since some of these factors change dynamically, it is useful for the provider’s policy to also exhibit flexibility (i.e. be changeable) as the system is running. It does not really matter which factors a provider uses to choose a transactional level, as long as the provider determines a level of transactional support it is willing to offer, ideally to the benefit of the provider.

4.1.2.2 Notifying The Client

Giving the provider the ability to decide on the level of transactional support it is willing to offer for a client request is only the first step in offering a service to the client. It is also necessary to inform the client of this decision. Thus it is important for the client to be notified of, and agree to, the level of transactional support being offered by the provider. This can be achieved by implementing an initial client enquiry, in which the client requests the level of transactional support that the provider is willing to offer for a given service
request. The provider must then inform the client of any transactional interaction patterns it is willing to provide for that service request. The client must then agree to one of these contracts before the service request can be handled in the required transactional manner. If the client does not agree with the initial patterns offered by the provider, it may wish to renegotiate with the provider until both sides have found a level of transactional support that they are willing to accept. Thus, when booking a hotel room, a client would first request the level of transaction support that the hotel is willing to offer for the request, and would only continue with the booking if the offered level was acceptable to the client.

4.2 Combining Contracts

When providers offer differing levels of transaction support it is necessary for clients who wish to utilise combinations of their services to combine the offered service levels in an acceptable manner. Typically, this will require a client to first ensure that there is a set of providers that will perform the actions necessary to successfully complete the client’s workflow with the level of transaction support required by the client. Since not every provider offers fully atomic transactions, there is possibly a risk for the client that at least part of an executing transaction may be completed without the possibility of that part being undone, or that a certain activity will have to be compensated, which may have an associated cost. However, as the actions necessary to complete a transaction can change based on results of earlier actions in the transaction, it is not always possible for clients to determine the level of risk to which they are exposed before starting the transaction.

4.2.1 The Client’s Transaction

Transactions in the Web environment are different to traditional transactions. In many ways, they are not transactions at all; instead, they are simply a workflow detailing the set of actions the client would like to succeed, and any alternatives that should be attempted in the case of failure. However, from the client’s point of view, it is a transaction: the entire action set is thought of as a single action, and, in the case that it cannot be completed successfully, the client would prefer it to seem that none of the actions were ever attempted. Consider a client who wishes to go to a concert in a nearby city. The client requires

---

Note that it is possible that the provider will be willing to offer more than one pattern; for example, a provider willing to offer the prepare/commit pattern will probably also be willing to offer the commit pattern, as the latter places less strain on the provider.
accommodation after the performance, and will not go to the concert if accommodation is not available. Further, the client would like a limousine to travel between the hotel and concert venue, but is willing to continue with the booking even if the car is unavailable. To the client, the booking of a hotel room, concert tickets, and a limousine are seen as a single action, even though the hotel, concert venue, and rental company have no other connection than through this client.

To define a transaction, the client must specify the workflow it wishes to complete. Essentially, this requires the client to specify which actions need to be completed for the client to accept that the transaction has completed successfully; any other case is seen as a failure of the transaction. Client workflows can be expressed in BPEL [88], or a similar language, as a structured activity that performs a set of actions either sequentially or in parallel. This set of actions can also include other nested structured activities, specifications as to whether any of the actions can be considered optional, and any alternative activities that should be attempted whenever a given action fails.

4.2.2 Initial Stages

When a client begins a transaction, it must first contact potential providers of the services it wishes to utilise. The client must then determine which providers it will use, and any risk that using such a provider will expose to the client. For example, a provider that offers a commit/compensate scheme may be preferred over one that offers a prepare/commit scheme if the client deems it unlikely that the rest of the transaction could succeed in the short time out period offered in the latter case. However, if there is a cost associated with the compensate step, the client must take this into account when determining if the transaction should continue.

In particular, when contacting service providers, the transactional offers of other providers may affect which levels the client is willing to accept from a provider, or how the workflow will proceed. For example, a client may be willing to accept a commit scheme for a pivotal action (such as the booking of a concert ticket) of their transaction if all of the other providers (such as the hotel) in the workflow offer a prepare/commit scheme, as the client can still complete the transaction without any risk by first preparing all other actions, and cancelling those actions only if the pivotal action fails. Similarly a client may choose to utilise an optional component in the workflow (such as
CHAPTER 4. DYNAMIC CONTRACTS FOR WEB SERVICES TRANSACTIONS

Algorithm 4.1 A client’s processing of a transaction.

1. Prioritise activities to be completed based on need and risk
2. While prioritised list not empty
3. If risk of performing next stage of first activity in list is acceptable
   4. Perform next stage of first activity in list
   5. Else
   6. If waiting is likely to alter this decision
   7. Wait
   8. Else if an alternate activity exists
   9. Replace activity with the alternate
  10. Else if activity is optional to the workflow
  11. Cancel the activity
   12. Else
   13. Cancel the workflow
  14. End if
  15. End if
16. Prioritise activities to be completed based on need and risk
17. End while

the booking of a limousine) if the provider offers the commit/compensate scheme, but may choose to omit it if only the commit scheme is offered.

4.2.3 Processing The Transaction

Once the client begins accepting agreements from various service providers, it must choose how to proceed. The general process a client should follow is listed in Algorithm 4.1. Essentially, the client should prioritise the tasks that are not yet completed and determine if the risk of performing the next step of the workflow is worthwhile. The next step will often be either a prepare or commit, but may also be a response to a time out or an enquire or compensate message. If the client deems it worthwhile, the next step of the highest priority action should be carried out. Otherwise the provider should either wait for the situation to change, or attempt to replace an activity with an alternative. If neither of these occur, the client must either cancel the activity if it is optional to the workflow, or cancel the entire workflow.

The main ambiguities in the described algorithm are:

- how the activities are prioritised
- how to determine whether a level of risk is acceptable; and
- how to determine when waiting is likely to be beneficial.
There is some information that can help with these decisions. The construction of the workflow can be used to determine the importance of various activities. If any provider offers an *enquire* stage, for example, then this can help to calculate the risk of attempting an action, as can whether a *prepare* or *compensate* stage is offered. A provider’s offer of callbacks may indicate that waiting is worthwhile. However, the actual decision on how these ambiguities are resolved must be reached by the client, and will depend on the workflow being attempted and the wishes of the client. Such issues will be examined more closely in Chapter 5.

Assuming that there is no cost for any action other than commit, and ignoring time outs, a client would act as follows:

1. The client should calculate the cost of performing all activities successfully.
   
   (a) If the cost is prohibitive, then any optional activities should be removed until the cost is acceptable.
   
     i. If all optional activities have been removed and the cost is still too high, the workflow must be abandoned.
   
     ii. Otherwise, the remaining activities in the workflow should be prioritised: those offering *prepare* first, then those offering *compensate*, and then all other activities.

2. The client should go through the prioritised list, preparing all that support a *prepare* stage.
   
   (a) If any fail when requested to prepare, they should be moved back in the prioritised list to a position after the actions that support compensation. This means that the client is waiting to process that activity later, but if any send a callback indicating that the prepare would now succeed, they should move again to high priority.

3. The client should then go through a similar process committing actions that support a *compensate* scheme.

4. Once this is done, if there are any actions that offer *prepare* or *compensate* stages that have not yet been prepared or committed, the client must decide whether to
risk having part of the workflow complete with those actions failing. This is the last stage at which the client can have the transaction fail with no risk.

(a) The client should send enquiries to any uncompleted actions that support such a query to help determine the risk of continuing.

(b) If any mandatory activity in the workflow fails the enquiry, then it is likely that the transaction will not complete successfully.

5. If the client determines that it is not worthwhile continuing, then all activities that have been committed can be compensated, and all that have been prepared can be cancelled, and it will appear as if the workflow had never started.

6. Otherwise, the client should start committing the remaining actions.

(a) If any fail then the client must again decide whether to continue. If the client decides not to continue, it can still compensate any committed activities that support compensation, and cancel any that have been prepared. However, any other actions that have been committed will remain completed.

7. If the client still wishes to continue the transaction once all of the remaining actions have been completed, it can commit all of the actions that have been prepared, and close the workflow. Provided no mandatory activity failed, the transaction will have completed successfully. Otherwise the client is at least willing to accept the result of the (possibly) partially-completed workflow.

4.3 Interaction Protocol

A well-defined protocol is required so that service providers and clients are able to both agree on the level of transaction support being offered, and ensure such a level is provided. This protocol must allow a provider to unambiguously define the transactional properties it is willing to offer for the requested service. This section describes a protocol to allow such an interaction.

Figure 4.1 shows the architecture of the dynamic transaction system. Initially, a client and provider use a negotiation protocol, described in Section 4.3.1, to mutually decide on the level of transaction support for a service call. This protocol is based on the
Figure 4.1: Dynamic Transaction System architecture.
CHAPTER 4. DYNAMIC CONTRACTS FOR WEB SERVICES TRANSACTIONS

Web Services Agreement Specification (WS-Agreement) [7] and involves the exchange of templates, the format of which are described in Section 4.3.2, to ensure that all parties agree on the level of support being offered. Once both the provider and the client wishing to utilise the provider’s service have agreed on a transactional contract, the transactional protocol of Section 4.3.3 is used to ensure that both parties behave correctly.

4.3.1 The Negotiation Protocol

The purpose of the negotiation protocol is to define the means by which the client and service provider agree on the level of transactional support that will be used for a given service call. Both parties must agree; the client may have an idea of the level(s) of transactional support it is willing to accept, and this is just as important as the level(s) that the provider is willing to offer. If no suitable agreement can be found, the client will not use the service offered by the provider; the client will have to either:

1. find an alternate provider that will offer an acceptable level of transaction support, or
2. modify or cancel the workflow it is attempting to complete

As a result of the second option, the provider will lose a client. Thus, there are incentives for both sides to reach a suitable agreement.

WS-Agreement allows for such an exchange. The specification includes:

1. a format to describe agreements and templates for allowing the negotiation of such agreements,
2. a protocol to allow two parties to establish an agreement, and
3. the ability to monitor agreements during runtime.

The protocol allows an Agreement Responder the ability to provide templates to an Agreement Initiator. These templates describe general agreements and specify which elements can be changed to create a concrete agreement, as well as constraints on those elements’ possible values. The Agreement Initiator can then use these templates to create an agreement request that the Agreement Responder will understand. Once the request has been sent, the Agreement Responder can decide whether to accept, or reject,
the request to create an agreement. In the case that the request is accepted, an agreement is created and both parties become bound to it.

The WS-Agreement specification provides a top-level structure for the description of agreements and agreement templates. Agreements in WS-Agreement begin with an optional name of the agreement, followed by contextual information such as which party is the service provider, or an optional expiration time for the agreement. The remainder of the agreement consists of terms which give details of any information or obligations of the parties to the agreement. These terms can be logically combined using AND/OR/XOR operations.

One of the most important components of an agreement are Service Description Terms, which give details of the offered or required functionality of services. The specification of this functionality uses a domain-specific language that can be independent of WS-Agreement. An agreement also contains Guarantee Terms to provide assurances and various other information about a service, such as quality of service or timing information. These guarantees have optional qualifying conditions to specify when the guarantee is valid. The guarantees also have service level objectives that can use a customised language to give a domain-specific description of the guarantee being offered. Finally, Guarantee Terms can also include a list of business values, which describe different value aspects of a particular objective. Essentially, these guarantees provide an assertion, by either one or both of the parties, about some aspect of the service being used. This assertion can include penalties or rewards that apply when the assertion is met or fails.

One of the problems with WS-Agreement is that there is no ability to renegotiate; the Agreement Initiator makes an offer, and the Agreement Responder can either accept or reject the offer. In many cases, it would be useful to allow an iterative process that allows offers to go back and forth until an agreement is made. One way to do this is to use WS-Agreement-Negotiation [129], which enhances WS-Agreement by adding a negotiateTemplate function that accepts one template as the input offer and returns zero or more templates as the counter offer. On receipt of a counter offer (which may just be the original input offer if that template is acceptable), the initiator determines whether any of the templates meet its requirements. If none do, another negotiation phase can be entered, or negotiations can cease. If, however, a template is acceptable, the initiator can use that template to create a new agreement.
Thus, WS-Agreement-Negotiation can be used to specify the level of transactional support offered by a provider by making the client behave as an Agreement Initiator, and the providers with which it wishes to interact as Agreement Responders. A language that describes the transactional properties of a service can then be used to specify Service Description and Guarantee Terms to ensure that both the client and the providers agree on the level of transactional support being offered for the service. Such a language is described in the next section.

4.3.2 Contract Format

As described above, WS-Agreement defines the basic format of a contract, but specific details can be included by utilising protocols specified by other languages to define terms. This section informally describes a protocol that can be used with WS-Agreement to provide terms that guarantee a level of transactional support for a client request. A more complete definition is provided in Appendix A.

4.3.2.1 Transaction Description Language

The Transaction Description Language (TDL) allows the specification, in an XML format, of how the five basic operations of enquire, prepare, commit, compensate, and callback are supported for a workflow, and can be embedded in a WS-Agreement contract or contract template. TDL defines a transaction by describing both the workflow to be performed, and the transactional properties that the provider must support for the workflow. The specification of complex workflows is beyond the scope of TDL, but it does support simple service calls, and includes extension points to allow other workflow description languages to be included, thus supporting an arbitrary level of complexity. More important for this language are the transaction properties TDL defines for the given workflow. An example TDL document is given below.
This example specifies that a hotel service accessible through http://www.example.com/bookRooms should be called with the parameter “1” (requesting to book one room), with the client first able to enquire and prepare the call before committing it, and with the ability to compensate after the commit has been completed.

The root element of a TDL document is <TransactionDefinition>. This has an optional id which can be used to identify the transaction, and contains a <TransactionDescription> child element. In turn, the <TransactionDescription> element contains the <WorkflowDefinition> and the <TransactionProperties> elements. Each of these elements can be extended by including child elements from other XML Schemas. The base language includes a <ServiceCall> element that can be placed in a <WorkflowDefinition> to identify a Web Service to call, and the parameters to pass to it.
The language defines four child elements for the `<TransactionProperties>` element. The first of these is `<Enquire>`. The `<Enquire>` element is used to indicate whether the `enquire` step is supported. It also has parameters to indicate whether `callback` is supported, both as a tentative hold and a call back if a failed request could now succeed. Next, the `<Prepare>` element can indicate that `prepare` is supported, and how long a successful request to prepare will be held before a time out occurs. The default action is for the workflow to commit after such a time out occurs, but this can be changed by setting the `cancelOnTimeout` attribute to `true`. The amount of time that a workflow will be held in the prepared state before a time out occurs is specified in the `duration` attribute. `<Prepare>` can also specify whether call backs are supported by setting the `callbackSupported` attribute to `true`, and the `callbackDuration` to specify an amount of time a callback remains valid after a failed `prepare`. Similarly, `<Commit>` has `callbackSupported` and `callbackDuration` attributes to determine call back behaviour at the `commit` stage. Finally, the `<Compensate>` element specifies how long the provider will stay in the committed state before completing the transaction (provided that compensation is supported), and whether the workflow should be completed or compensated in the event such a time out occurs.

### 4.3.2.2 Contract Format Example

This section examines the following example contract template.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsag:Template
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"
    xmlns:tdl="http://davidjohnpaul.com/transaction/2010/tdl"
http://schemas.ggf.org/graap/2007/03/ws-agreement agreement_types.xsd"
    wsag:TemplateId="HotelBookingExample1">

  <wsag:Name>RoomOffer</wsag:Name>

  <wsag:Context>
    <wsag:ServiceProvider>wsag:AgreementResponder</wsag:ServiceProvider>
  </wsag:Context>

  <wsag:Terms>
```
<wsag:All>
  <wsag:ServiceDescriptionTerm wsag:Name="Room Booking TDL"
    wsag:ServiceName="http://www.example.com/bookRooms">
    <tdl:TransactionDefinition id="RoomBookingTransaction">
      <TransactionDescription>
        <WorkflowDefinition>
          <ServiceCall name="http://www.example.com/bookRooms">
            <ServiceParameter type="string">1</ServiceParameter>
          </ServiceCall>
          <WorkflowDefinition>
            <TransactionProperties>
              <Enquire supported="true"
                callbackSupported="false"
                tentativeHoldSupported="false"/>
              <Prepare supported="true" duration="P1H"
                callbackSupported="false" cancelOnTimeout="false"/>
              <Commit callbackSupported="false"/>
              <Compensate supported="true"
                commitDuration="P06H"
                cancelOnTimeout="true"/>
            </TransactionProperties>
            <TransactionDescription>
          </tdl:TransactionDefinition>
        </tdl:TransactionDefinition>
      </TransactionDescription>
    </wsag:ServiceDescriptionTerm>
  </wsag:All>
</wsag:Terms>

<wsag:CreationConstraints>
  <wsag:Item Name="NumRooms">
    <wsag:Location>@tdl:ServiceParameter</wsag:Location>
    <wsag:ItemConstraint>
      <xs:restriction base="xs:positiveInteger">
        <xs:maxInclusive value="10"/>
      </xs:restriction>
    </wsag:ItemConstraint>
  </wsag:Item>

  <wsag:Item Name="EnquireSupported">
    <wsag:Location>@tdl:Enquire</wsag:Location>
    <wsag:ItemConstraint>
      <xs:restriction base="xs:boolean"/>
    </wsag:ItemConstraint>
  </wsag:Item>

  <wsag:Item Name="PrepareSupported">
    <wsag:Location>@tdl:Prepare</wsag:Location>
    <wsag:ItemConstraint>
      <xs:restriction base="xs:boolean"/>
    </wsag:ItemConstraint>
  </wsag:Item>
</wsag:CreationConstraints>
CHAPTER 4. DYNAMIC CONTRACTS FOR WEB SERVICES TRANSACTIONS

The template begins with the WS-Agreement Template element, which defines the unique TemplateId to be “HotelBookingExample1”. After this, the template is allocated the (not necessarily unique) name “RoomOffer”. The Context element then specifies which party is the service provider. Following this are the terms, which describe the actual service that will be provided. In this case, it is a TDL document that books rooms from the http://www.example.com/bookRooms service. The TDL description passes a default parameter of “1” to the service (booking a single room), and offers support for enquire, prepare (with a default time out of one hour), commit, and compensate (with a default time out of six hours).

Any contract matching this template will utilise a TDL document similar to that in the WS-Agreement Terms of the template. The CreationConstraints after the Terms specify...
where an actual agreement can differ from the template. The first item specifies a range for the parameter passed to the *bookRooms* service. The range is specified to be a positive integer no greater than 10, meaning that up to ten rooms can be booked with a single service call. The next three items respectively allow support for enquire, prepare, and compensate to be enabled or disabled for the service call. Finally, the last two items allow a change of duration for the time outs periods applied to the prepare and commit operations. This allows them to be extended beyond the default times given, though does place some restrictions on possible durations. For example, the prepare time out cannot be longer than nine hours, as the pattern its duration must match, “P\p{Nd}{1}H”, specifies that only a single digit number of hours is supported.

### 4.3.3 The Transaction Protocol

Once a client starts to reach agreements with providers, it can begin determining whether or not the transaction should continue. In particular, the client can determine whether it should start transaction processing with the individual providers with which it is communicating. For example, the client wishing to go to the concert may immediately *prepare* the room booking if the hotel offers a risk-free *prepare/commit* scheme. Alternately, if the *prepare* step has an associated cost, or no *prepare* or *compensate* step is available, the client may have to wait on replies from the concert venue and limousine service before deciding it is willing to continue with the accommodation booking.

When a client does decide to continue interaction with a provider, the way it does so depends on the level of transaction support being offered. The specification described here is based on the *MixedOutcome* Business Activity coordination type included in WS-BusinessActivity [92]. In particular, it inherits security and fault handling schemes from this standard. To allow compatibility with this standard, for the remainder of this section the client will be called the *coordinator* and the provider will be called the *participant*. This section only describes the interaction that occurs when the provider offers an *Enquire/Prepare/Commit/Compensate* scheme with coordinator (i.e. client) completion, presumed commit, and presumed compensate, as this scheme includes many of the details used in the other schemes. For explicit specification of the other possible schemes, see Appendix B.

The state diagram in Figure 4.2 illustrates the abstract behaviour for a coordinator
Figure 4.2: EnquirePreparePresumedCommitPresumedCompensate abstract state diagram. Solid lines represent coordinator-generated messages. Dashed lines represent participant-generated messages.

running a transaction and a participant providing a service for use in the transaction. This diagram shows the states that the coordinator and participant go through to determine their view of the protocol’s progress. The message sender’s view changes as soon as it sends the message, and the receiver’s as soon as it receives a message. Since messages are not sent instantaneously, the respective views of a coordinator and participant may be inconsistent for short periods of time. For full details of how these and other protocol intricacies are handled, see Appendix B.

4.3.3.1 Participant Messages

The messages accepted by the coordinator are:
4.3.3.1 cannotComplete

This message indicates that the participant has determined it cannot successfully complete all processing related to the protocol instance. The participant has reverted to a state in which no work for this protocol instance has taken place, and will not interact with the activity any more, other than accepting the notCompleted reply from the coordinator.

4.3.3.1.2 enquiryFailed

Receipt of this message indicates to the coordinator that, at the time of sending the message, the participant would not have been able to successfully complete the operation that the coordinator is requesting of the participant. This message can decrease the coordinator’s confidence that the operation will finish successfully, without providing any guarantee.

4.3.3.1.3 enquirySuccessful

Receipt of this message indicates to the coordinator that, at the time of sending the message, the participant would have been able to successfully complete the operation that the coordinator is requesting of the participant. This message can increase the coordinator’s confidence that the operation will finish successfully, without providing any guarantee.

4.3.3.1.4 exit

This message indicates that the participant will no longer participate in the current activity. The participant has reverted to a state in which no work for this protocol instance has taken place and will not interact with the activity any more, other than accepting the exited reply from the coordinator.

4.3.3.1.5 fail

Upon receipt of this message, the coordinator knows that the participant has failed. Receipt of this message from the Active state indicates that no work has taken place. Receipt of this message from the Committing or Compensating states indicates that the participant was unable to fulfil a promise it made to the coordinator, and the state of work is undetermined.
CHAPTER 4. DYNAMIC CONTRACTS FOR WEB SERVICES TRANSACTIONS

4.3.3.1.6 prepared

When the coordinator receives this message, it is guaranteed that the requested operation will succeed. Unless the coordinator cancels the operation, or a timeout occurs, the coordinator knows that sending a commit message will result in the operation completing successfully.

4.3.3.1.7 committed

This message indicates to the coordinator that the operation it is requesting has been performed. Unless the coordinator compensates the operation, or a timeout occurs, the coordinator knows that the operation has completed successfully.

4.3.3.1.8 timeOut

On receipt of this message, the coordinator knows that a timeout has occurred, and the default action of the protocol will occur.

4.3.3.1.9 cancelled

Once this message is sent, the participant should forget about the activity. The coordinator has cancelled all processing, so the participant should behave as if this activity had never been started.

4.3.3.1.10 compensated

Once this message is sent, the participant should forget about the activity. The activity has been compensated, with any required actions to achieve this result being completed successfully.

4.3.3.1.11 closed

Once this message is sent, the participant should forget about the activity. The activity has been completed successfully, and can no longer be undone.

4.3.3.2 Coordinator Messages

The messages accepted by the participant are:

83
4.3.3.2.1 enquire

On receipt of this message, the participant checks to see whether the coordinator’s request could currently be completed successfully. If it could, the participant may reply with enquirySuccessful. Otherwise, the participant must reply with enquiryFailed.

4.3.3.2.2 prepare

On receipt of this message, the participant should attempt to ensure that the operation requested by the coordinator can be completed. If the operation will be guaranteed to successfully complete, the participant must reply with prepared. Otherwise, the participant must reply with fail.

4.3.3.2.3 commit

On receipt of this message, the participant should perform the operation requested by the coordinator. If this is done successfully, the participant must reply with committed. Otherwise, the participant must reply with fail.

4.3.3.2.4 close

On receipt of this message, the participant knows that the activity has completed successfully. Replying with the closed message ends the protocol instance.

4.3.3.2.5 exited

On receipt of this message, the participant should forget about this activity. The activity has been exited before any work has taken place, and both parties should behave as if the interaction never occurred.

4.3.3.2.6 compensate

Upon receipt of this message, the participant knows it must compensate the work it has performed as part of this protocol instance. If the work is successfully compensated, the participant should reply with a compensated message. Otherwise the participant should reply with a fail message.
4.3.3.2.7  notCompleted

Upon receipt of this message, the participant knows the coordinator is aware that the participant is unable to complete the protocol. Both parties should forget about the activity.

4.3.3.2.8  failed

On receipt of this message, the participant knows the coordinator is aware of the failure of the participant to complete a required stage. The protocol is completed, but the failure may indicate that the participant has breached the contract.

4.3.3.2.9  cancel

On receipt of this message, the participant knows the coordinator no longer wishes to perform the activity. After sending the cancelled reply, the participant should forget about the activity.

4.3.3.2.10  timedOut

Receipt of this message indicates that the coordinator has acknowledged the timeout and processing should continue.

4.3.4  Time Outs

One of the weaknesses of the WS-BusinessActivity is its limited support for time outs [119]. The WS-BusinessActivity standard essentially follows a Commit/Compensate pattern as described above, without the ability for the participant to send a timeOut message from the Committed state. Instead, a business activity can optionally specify an expiry date, after which either the coordinator or the participant can choose to leave the activity simply because it is taking too long. However, the participant cannot leave the activity if it has already committed; it must wait for the coordinator to send a close or compensate message before it can finish the interaction. This can be unfair for the provider, especially if the connection with the coordinator is broken, as, in this circumstance, the provider must maintain the ability to compensate the activity indefinitely.

The time outs suggested above solve this problem, but create a similar problem for the coordinator. A malicious participant could ignore a message from a coordinator and
instead send a time out. Consider if the time out action was to compensate an activity and the coordinator sent a close message. The participant could pretend that it did not receive the close message and instead reply with a timeOut, forcing the coordinator into a Compensating state; this would be especially unfair if the compensating action had an associated cost for the coordinator.

Fortunately, it is possible to combine other Web Services standards to avoid these problems. By using WS-Security [71] to include timestamps with messages and WS-ReliableMessaging [90] to allow both parties to send sequenced messages with acknowledgement that they have been received, both coordinator and participant can have guarantees about message order.

### 4.4 Linking With Existing Services

To facilitate real-world use, it is beneficial to allow support for existing services that do not support the dynamic negotiation of transactional support described in this chapter. Section 4.4.1 describes a technique to allow such backwards compatibility.

#### 4.4.1 Backwards Compatibility

Abstract services [105] can be used with the dynamic transaction scheme described in this chapter. Rather than directly communicating with the various service providers, the client contacts an abstract service that keeps an up-to-date list (using any available service discovery mechanisms) of the providers offering a given service. Thus, a client’s initial request is handled by the abstract service, which then contacts the various service providers to see which (if any) are willing to provide the service to the client. The abstract service then returns to the client the set of transactional templates offered by each of the providers that are willing to accept the request.

If the client accepts one of those contracts then the abstract service facilitates the service call by acting as a proxy between the client and the provider that offered the accepted contract. If the client does not accept any of the returned contracts and wishes to negotiate, then it passes its negotiation messages to the abstract service, which then passes them on to all the service providers that were willing to perform the action, again returning any responses to the client. Thus, all communication between the client and the
service providers is done through the abstract service.

These abstract services make it possible to include providers that do not utilise the transactional scheme described in this system. An abstract service that knows the level of transaction support offered by a non-supported provider can supply an interface that transparently converts a client’s request to the request required by the actual provider. Thus, if a client requests a contract from the abstract service, it returns a non-negotiable contract that specifies the actual provider’s level of support. If the client chooses that contract (rather than one of the other contracts offered by the abstract service), then the client interacts with the abstract service using the dynamic transaction system, and the abstract service converts the client requests to those required by the actual provider to allow the required transaction support. The abstract service converts any replies from the non-supported provider into the response expected by the client, leaving the client unaware that the service it is using does not support the dynamic transaction system.

4.5 Evaluation

The dynamic transaction system introduced in this chapter offers greater flexibility than that available in current standards. In fact, this system can be seen as a superset of currently available systems, as demonstrated by the backwards compatibility described in Section 4.4.1. In previous systems, if ever a provider wanted to alter the level of transaction support it offered with a service, it would have to define a new service with the new level of transaction support. Each client would then have to discover the altered service before they were able to use that new service. The dynamic transaction system, on the other hand, offers a single contact point for the service, and the transaction level can be altered without requiring changes to each client’s service identifier. This allows the service provider to modify the level of transaction support it offers based on its current state.

The main drawback of the dynamic transaction scheme is that it requires the client to first determine the level of transaction support that the provider is willing to offer before the actual interaction can begin. This requires the exchange of at least three messages between the client and the provider, which is really unnecessary for any service that never alters the level of transaction support it offers. However, each of these messages is small.
in size, so the overhead would not be a great burden for any service that may later require the increased flexibility that the dynamic transaction system provides.

4.6 Summary

This chapter introduced a system to allow the dynamic negotiation of the level of transactional support offered to clients by service providers. The first stage was to identify the possible transaction patterns a service provider can offer, which were described through the five basic operations: *enquire, prepare, commit, compensate*, and *callback*. This led to the realisation that providers may wish to vary the support they offer for a particular service as the providers’ circumstances change. To allow this dynamic support, it is necessary for a provider to implement some way to notify a client of the transaction support it wishes to offer for a particular service call.

Since clients may wish to combine services that have different transactional support into a single workflow, it must be possible for a client to reason about the level of risk afforded by a particular set of transactional guarantees. Since not all interactions allow for fully ACID transactions, there is a chance that the client may risk having part of their workflow succeed, or may have to pay compensation to undo operations that have already been completed if the overall workflow fails. It is up to the client to determine whether or not that risk is acceptable. The client can then combine the contracts offered by the providers to minimise the client’s risk, and achieve as close to a fully ACID transaction as possible.

To help alleviate the client’s risk, a technique is presented to allow negotiation of the offered transactional guarantee with a provider. This allows the provider to modify the level of transactional support it is willing to offer a client based on issues such as the client’s needs. These negotiations can allow a client to begin an interaction that it would otherwise find too risky, which can benefit both the client (because it can attempt its workflow) and the provider (by giving it an extra client).

Abstract services can also be incorporated into this system. These services provide a single point of contact for a client to interact with multiple different providers. The provider of an abstract service may offer additional support for a client, such as combining actual services from multiple providers into a single service for the client. Further, abstract
services can allow backwards compatibility by defining the levels of transaction support offered by services that do not support the dynamic transaction system and offering a compatible interface to allow clients to utilise the legacy service as if it did support dynamic transactions.

The chapter described the dynamic transaction system at a high level. This included techniques to incorporate the dynamic transaction system with existing standards to allow real-world use. The next chapter will examine the fundamentals of the system and provide formal reasoning to show how the system can be used to guarantee correct transactional support. The formal model will lay the foundation for development of a simulator for modelling dynamic transactions.
Chapter 5

A Formal Model Of The Dynamic Transaction System

Thought and theory must precede all salutary action.

William Wordsworth

The previous chapter introduced a dynamic transaction system that allows a provider to alter the level of transaction support it offers to clients as the provider’s state changes. This chapter examines the system in more depth, providing a formal model to ensure correct behaviour of all involved participants. The formal system allows automatic reasoning by participants, removing much of the ambiguity and uncertainty in how such a system can be used.

While providing a mechanism to allow automatic reasoning, it is also important that all participants in the system retain as much freedom of choice as possible. All providers and clients may wish to be completely autonomous, but cooperation necessitates a surrender of some of this autonomy. The presented model reduces the amount of autonomy that must be surrendered as much as is practical. When offering or using particular levels of transactional support, providers and clients do have to ensure that certain restrictions of their freedom are met, but, in each case, that is a decision individual participants must make before offering or accepting the transactional guarantees.

It is vitally important that a provider can support any transactional guarantee it offers to a client. Without such assurances, the provider cannot be trusted, and is likely to cause a breach of contract. Providers must therefore track the various resources they are
offering and any promises they have made in relation to those resources. When offering a new transactional guarantee, a provider must be certain that the level of support it is offering is compatible with the guarantees it has given to other clients. Section 5.1 uses a state-transition-based approach to allow providers to track the guarantees it has offered and allow such assurances to be made.

Clients have different decisions to make. Given a workflow, the client may be offered a different transactional guarantee for each of the actions in that workflow. The client must be able to ensure that, when the guarantees are combined, the workflow ends in an acceptable state. In particular, a client should cancel its transaction if ever it determines that it may find the results unacceptable. Through use of a model in which a client specifies budgets for the workflows it wishes to complete, it is possible for the client to determine whether attempting the next action required for its workflow represents an acceptable risk, or whether the client is better off cancelling its actions. This is explored in Section 5.2.

Given these models, both providers and clients are better able to determine a negotiation strategy for determining the level of transaction support to offer or accept. While an in depth investigation of negotiation strategies is outside the scope of this work, Section 5.3 examines how various properties of the formal systems can be used by providers to determine what levels of transaction support to offer, and by clients to determine which levels to accept. This can then be extended to allow both parties to determine when to negotiate, and any concessions that the various parties can afford to give.

5.1 Offering Dynamic Transaction Support

It is extremely important that a service provider can honour any transactional guarantee it offers. If it cannot, this could cause a breach of contract or, at the very least, reduce the level of trust that clients have in the provider. In order to ensure that transactional guarantees can be kept, it is sufficient that the provider to track the state of each resource it is offering, and only offer guarantees that are compatible with the current state of its resources. Section 5.1.1 describes the various states in which resources offered by a provider can be, and how different transactional guarantees affect transitions between these states. Section 5.1.2 then specifies how the current state of the resources offered by a provider can
be used to determine the possible levels of transaction support the provider can honour. Through such examination, a provider can ensure that it only offers new clients those levels of transaction support that it can guarantee.

5.1.1 Tracking The Current State Of A Provider’s Resources

In order to know the levels of support that a provider can offer for a particular resource, it is important for the provider to know what promises it has already made for that resource. For example, if one client has sent a request to commit an operation that utilises a particular resource, then the provider cannot offer that same resource to any other client, unless the first client later compensates the commit action. Section 5.1.1.1 lists the possible states in which a provider’s resource can be, and Section 5.1.1.2 examines how different client actions move resources between states.

5.1.1.1 Possible States

When a provider is offering a certain number of a particular resource, each of the individual resources can be in one of four possible states:

**Free** the provider has offered no guarantees for the resource and it is freely available.

**Locked** the provider has promised the resource to a particular participant, but that participant has not yet booked the resource. The resource is unavailable, but the participant may end up not making the booking. This is the case when the provider has offered the prepare operation and the client has prepared but not yet committed the activity.

**Booked** a participant has booked the resource, making it unavailable, but may later cancel the booking. This is the case when the provider has offered the compensate operation and the client has committed but not yet closed the activity.

**Finalised** a participant has booked the resource and is unable to cancel that booking. The resource is unavailable, as it belongs to the participant who booked it.

Each individual resource begins in the **Free** state, and the resource’s current state depends on the transactional guarantees given by the provider, and the current progress through interactions with clients. In many cases, the provider will not need to differentiate between
the Locked and Booked states, but there is a difference between the two: in the Locked state, the resource still belongs to the provider, while in the Booked state, the resource belongs to the participant who has made the request.

5.1.1.2 State Transitions

The current state of a provider’s resources depends on the progress of the provider’s interaction with clients requesting the resources. The provider can offer different levels of transaction support using a combination of the Enquire, Prepare, Commit, Compensate, and Callback operations described in the previous chapter. However, any callback operations in the patterns supported by the provider can be considered as notifications of the change of a resource’s state. For example, when a tentative hold is offered, a notification is sent when the resource that the hold is on moves from the Free state to one of the other states. Similarly, all other callback operations only mark a resource’s transition to or from the Free state. Further, the Enquire operation is used only to let the requesting participant know whether the provider’s resources are currently available, without providing any guarantee or changing the state of any of the provider’s resources.

Thus, when determining state transition rules for a provider’s view of its resources, it is enough to consider only the Prepare, Commit, and Compensate operations.
Figure 5.1 describes the state transitions necessary when a provider agrees to support a particular action. From the Free state, if the provider agrees to a Prepare operation, then the requested resource moves to the Locked state. Otherwise, a resource only moves from the Free state when the provider agrees to Commit a booking. From both the Free and Locked state, an agreement to Commit moves a resource to the Booked state whenever the interaction being used supports the Compensate action. Otherwise, Commit moves the resource to the Finalised state. Similarly, when a Close message is sent, the resource moves from the Booked state to the Finalised state. In either the Locked or Booked state, an agreement to Cancel moves the resource back into the Free state, making it available for the provider to offer to other participants.

5.1.2 Determining The Available Levels Of Transaction Support

A provider can use the current state information of its resources to determine whether a request in a transactional interaction pattern to which it has agreed should succeed or fail. This places no real restriction on the levels of support that a provider should offer; a provider with no resources available may still offer a Prepare/Commit transaction pattern, but will be required to refuse the first call to prepare because the resources are unavailable. However it is typically better for the provider to only offer levels that it has the resources to support, giving a requesting participant some indication that the transaction pattern being offered has a chance of success. Thus, while it is largely a matter of choice for the provider as to what transaction patterns to offer for a particular request, the current state of the provider’s resources does give some indication as to what levels should be offered.

The current state of a provider’s resources can be used to guide the levels of support that should be offered, or those that should be avoided. For example, if a provider does not currently have enough resources in the Free state to support a client’s request, but believes that it may in the future, then offering a transactional pattern that includes Enquire or Callback may be more beneficial than when enough resources are currently in the Free state and an immediate Prepare or Commit action is more likely to succeed. For another example, a provider should only offer a transaction scheme with Prepare or Commit as its first action if it believes that a request to prepare or commit the request would be successful; the provider can determine this by ensuring that it has enough Free resources to support the request.
CHAPTER 5. A FORMAL MODEL OF THE DYNAMIC TRANSACTION SYSTEM

5.2 Using Transactional Guarantees

A client workflow is really only transactional because that is how the client views it. Each provider offers a transactional guarantee for a single action, but a client workflow may combine multiple different actions from many different providers, each with a different transactional guarantee. It is important that the client can combine the different guarantees and reason about the overall workflow to determine whether the client will find the outcome of its transaction acceptable. Using this ability, the client can cancel any transaction that may end in an unacceptable state to ensure that the client is satisfied with the end result.

A client’s transaction can be arbitrarily complex. A workflow contains a set of actions, where each action is either a single service call or another workflow. Each workflow may contain an alternate workflow which is attempted whenever the main workflow fails; a workflow is considered successful if either it completes successfully or its alternate workflow completes successfully. Parts of a workflow can be considered optional, where their success is not required but is preferred, or compulsory where the success of the workflow depends on the successful completion of those actions. This nesting of workflows and differentiation between optional and compulsory components can make it difficult for a client to reason about the progress of the transaction.

Since some transactions, in the case of a failure, finish with the client worse off than when it started, it is in the client’s best interest to detect that a workflow may fail as early as possible. Further, the client must monitor the workflow to ensure that, regardless of the outcome, the workflow ends in a state the client is willing to accept. To this end, an automatic system to allow clients to reason about their workflows is beneficial. This section presents a strategy to allow such automatic decisions.

The strategy presented in this thesis relies on the concept of client budgets. Each client has a maximum budget it is willing to spend to ensure the successful completion of its workflow, and a maximum budget that must not be exceeded should the workflow fail. The client then continually calculates the set of actions that are possible from the current state of its workflow and, for each possible action, uses the contracts offered by the service providers to calculate the minimal cost to complete the workflow successfully and the maximal cost to complete the workflow unsuccessfully. By ensuring that the costs
of any action performed do not exceed the client’s budgets, the client can guarantee that
the workflow will only end in a successful state if the client’s success budget has not been
exceeded, or, if the workflow fails, that the overall cost of the workflow is below the client’s
fail budget. As the client only specifies budgets which it is willing to spend, this ensures
that the client ends in a state it finds acceptable.

Section 5.2.1 introduces notation to describe the current state of a client’s workflow,
which is used in Section 5.2.2 to determine the next action that a client should perform.
The strategy presented in those sections assumes the client has complete knowledge of
the contracts being offered by each provider involved with the workflow. In Section 5.2.3,
however, the strategy is extended to support client reasoning when some of the contract
offers have not yet been received. It is assumed that, when a client does receive a contract,
that contract is the best that the client will receive for the given service. In other words,
negotiation of the contract has already been completed before the contract is used to
calculate a client’s next action. While complete negotiation strategies are beyond the
scope of this thesis, some details of a client’s contract negotiation are detailed in Section
5.3.2.

5.2.1 Describing A Client’s Workflow

A client’s workflow combines multiple single interactions with (potentially) numerous
providers into a complex structure that defines the order in which services should be
performed, how the failure of particular services or groups of services should be handled,
and what constitutes a successful completion of the workflow. In order for a client to
make an informed decision as to how it should proceed with the workflow, it is necessary
that the client has a detailed understanding of the current state of the workflow. This
section details how a client can track the state of, firstly, a single interaction with a
single provider in Section 5.2.1.1, and then an entire workflow in Section 5.2.1.2.

The notation used to describe a workflow is based on $\pi$-calculus [82], in particular
$\pi t$-calculus [16], which is a process calculus used to describe interactions between different
processes. The work in this thesis borrows from the algebraic laws of the calculus to allow
a client to reason about the workflow it is attempting. These algebraic laws are augmented
with client reasoning about the individual interactions included in its combined workflow.
Thus, while based on $\pi$-calculus, the system presented here is not a process calculus, but
instead a tool to allow client reasoning. In particular, a client only requires a model of its own workflow and a knowledge of the required transactional behaviour of the providers it is using to allow its reasoning, whereas most process calculi require a complete description of not only all service providers that may be used by the client, but also of any other clients that may utilise those provider’s services while the first client’s workflow is still processing [70]. By thus reducing the complexity of the model, it is possible for a client to guarantee its correct behaviour, rather than having the system grow to such a size that such analysis become intractable (as in [44]).

5.2.1.1 Describing A Single Interaction

Client workflows are comprised of single interactions in which the client requests a particular service from a particular provider. Using the dynamic transaction system introduced in Chapter 4, such an interaction transitions through a series of states based on the level of transaction support being offered for the service call. In fact, by using cost as an edge weight, and giving the unsupported operations (of the five transaction operations: enquire, prepare, commit, compensate, callback) a cost of $\infty$, it is possible to use a single state transition diagram to describe all possible interactions (regardless of the level of transaction support offered), with only the edge weights changing, based on the contract offered by the provider performing the service.

Figure 5.2 is a state transition diagram that shows the client-side view of an interaction with a provider. The states presented in this diagram are described in Section 5.2.1.1.1, and the form of the contract which describes the edge weights is in Section 5.2.1.1.2. Using a contract and the current state through the interaction, it is possible to describe the client’s progress through the interaction as specified in Section 5.2.1.1.3.

5.2.1.1.1 Possible States

Regardless of the transactional guarantee offered by a provider, a client can see its interaction with a provider to be in one of the following states:

**Initial** The starting state for an interaction, indicating that a contract has been offered to the client but has not yet been accepted.

**Active** The state in which the contract that was offered by the provider has been
CHAPTER 5. A FORMAL MODEL OF THE DYNAMIC TRANSACTION SYSTEM

Figure 5.2: State transition diagram for a client’s processing of an activity. Solid lines represent messages sent by the client. Dashed lines represent messages sent by the service provider. Dotted lines indicate client-side only transitions. Values in parentheses represent the associated cost to the client.
accepted by the client.

**Enquired** The state in which a client has sent an enquiry to have the provider indicate whether the resources required for the successful completion of the interaction are currently available, but has not yet received a reply.

**EnquiryFailed** The state in which the provider has replied that the resources required for the successful completion of the interaction were not available when an enquiry request was received from the client.

**EnquirySuccessful** The state in which the provider has replied that the resources required for the successful completion of the interaction were currently available when an enquiry request was received from the client.

**Preparing** The state in which the client has sent a request to prepare the operation required by the interaction but has not received a reply as to whether the request was successful.

**NotPrepared** The state in which the provider has indicated that a request from the client to prepare the operation required by the interaction was unsuccessful.

**PrepareCallback** The state in which the provider has indicated that a request from the client to prepare the operation required by the interaction that was unsuccessful would now succeed.

**Prepared** The state in which the provider has indicated that a request from the client to prepare the operation required by the interaction was successful.

**Committing** The state in which the client has sent a request to commit the operation required by the interaction but has not yet received a reply as to whether the request was successful.

**NotCommitted** The state in which the provider has indicated that a request from the client to commit the operation required by the interaction was unsuccessful.

**CommitCallback** The state in which the provider has indicated that a request from the client to commit the operation required by the interaction that was unsuccessful would now succeed.
Committed The state in which the provider has indicated that a request from the client to commit the operation required by the interaction was successful.

Compensating The state in which the client has sent a request to compensate the operation that has been committed.

Successful The state indicating that the interaction has completed with the requested operation being completed successfully.

Failed The state indicating that the interaction has completed without the requested operation being completed successfully.

Let us label this set of states $S$. An interaction begins in the Initial state, and transitions between states occur as messages are sent between the client and the provider. From the Initial state, as seen in Figure 5.2, the client can choose to accept or reject the contract. If the client rejects the contract, then the interaction transitions to the Failed state. Otherwise the interaction moves into the Active state. From the Active state, either party can cancel the interaction, moving it to the Failed state. Other options from the active state allow the client to request either enquire, prepare, or commit operations.

In the case that an enquire request is sent, the provider may reply that the request was successful (indicating that the provider can currently successfully complete the activity), which moves the interaction to the EnquirySuccessful state. If, on the other hand, the provider replies that the enquiry is unsuccessful (indicating that the provider cannot currently complete the required activity successfully), the interaction moves to the EnquiryFailed state. From either of these states, the client can initiate a new enquiry, though any enquiry callbacks remove the need as they transition the client between the EnquiryFailed and EnquirySuccessful states without requiring a new request. Similarly to the Active state, either party can cancel the interaction from both the EnquiryFailed and EnquirySuccessful states, and the client can request the prepare and commit operations.

When a prepare request is sent, the provider can either send a cannotComplete reply to indicate that the prepare has not occurred, or a prepared message to indicate that the prepare has been performed successfully. On receipt of a cannotComplete message, the client can either acknowledge the failure, moving to the Failed state, or, if applicable, wait for a call back from the provider and then send another prepare request. If the prepare
is completed successfully, the client can choose to either cancel the activity, or request to have it committed.

After a request to commit, the provider can either perform the requested service successfully, or indicate that it cannot be completed at the current time. If the commit request was sent from the Prepared state then the second option should not be possible. On receipt of a committed message, the interaction moves to the Committed state. With compensation support, the client can choose to either close the interaction, moving it to the Successful state, or compensate the activity, moving to the Failed state. If the provider indicated that it could not complete the commit operation then the interaction moves to the Failed state unless the provider sends a callback to the client to indicate that the commit operation would now succeed.

Finally, from the Successful state, the client may choose, without the provider’s knowledge, to take some action to undo the operation performed by the provider. In this case, the provider would believe that the interaction completed successfully, while the client would see it as a failure. This allows the client to compensate the action performed by the provider even when the provider does not support compensation. For example, if the client is requesting an item from the provider that is shipped to the client, the client could immediately dispose of the item to effectively have it appear as though the operation failed.

5.2.1.1.2 Describing A Contract

In order to determine the edge weights for the transitions described in Figure 5.2, it is necessary for the client to have a contract from the provider offering the service. The contract describes which of the basic transactional operations are supported for the service call, their costs, and any time limits associated with any of the actions (see Section 4.3.2.2 for an example). For the purpose of this model, it is assumed that time is a discrete quantity represented by an integer.

A contract \( c \in C \) is a tuple \( (oto, enq, ecb, eth, pre, pcb, pto, com, ccb, cto, cmp) \) where:

\( oto \in \mathbb{N} \) is the length of time after the contract is offered for which the offer is valid. Once the specified amount of time has passed, the contract will no longer be accepted by the provider.
CHAPTER 5. A FORMAL MODEL OF THE DYNAMIC TRANSACTION SYSTEM

\(enq \in \mathbb{R}\) is the cost of an enquire operation. A value of \(\infty\) indicates that the operation is not supported.

\(ecb \in \{-1\} \cup \mathbb{N}\) is the length of time after a call back after a failed enquiry that the original enquire operation is guaranteed to succeed. A value of \(-1\) indicates that this call back is not supported. 0 indicates that no future guarantee is given.

\(eth \in \{-1\} \cup \mathbb{N}\) is the length of time after a call back after a successful enquiry that the original enquire operation is guaranteed to succeed. A value of \(-1\) indicates that this call back is not supported. 0 indicates that no future guarantee is given.

\(pre \in \mathbb{R}\) is the cost of a successful prepare operation. A value of \(\infty\) indicates that the operation is not supported.

\(pcb \in \{-1\} \cup \mathbb{N}\) is the length of time after a failed prepare that a call back guarantees the original prepare operation will succeed. A value of \(-1\) indicates that this call back is not supported. \(\infty\) indicates that no time out is enforced and the guarantee will only expire after the receipt of a cancel or prepare message.

\(pto \in \mathbb{Z}\) is the length of time that can pass after a successful prepare without a request to commit or cancel before the default action occurs. When \(pto < 0\), the action taken after a time out is to cancel the activity. When \(pto \geq 0\), the action taken after a time out is to perform the commit action. In either case, \(|pto|\) is the length of time before the time out occurs. When \(|pto| = \infty\), no time out occurs.

\(com \in \mathbb{R}\) is the cost of a successful commit action.

\(ccb \in \{-1\} \cup \mathbb{N}\) is the length of time after a failed commit that a call back guarantees the original commit operation will succeed. A value of \(-1\) indicates that this call back is not supported. \(\infty\) indicates that no time out is enforced and the guarantee will only expire after the receipt of a cancel or commit message.

\(cto \in \mathbb{Z}\) is the length of time after a successful commit that can pass without a request to close or compensate before the default action occurs. When \(cto < 0\), the action taken after a time out is to compensate the activity. When \(cto \geq 0\), the action taken after a time out is to close the activity. In either case, \(|cto|\) is the length of time before the time out occurs. When \(|cto| = \infty\), no time out occurs.
CHAPTER 5. A FORMAL MODEL OF THE DYNAMIC TRANSACTION SYSTEM

<table>
<thead>
<tr>
<th>From State</th>
<th>Message</th>
<th>To State</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enquired</td>
<td>enquiryFailed</td>
<td>EnquiryFailed</td>
<td>$\infty$ if $ecb = -1$ otherwise $0$</td>
</tr>
<tr>
<td>Enquired</td>
<td>enquirySuccessful</td>
<td>EnquirySuccessful</td>
<td>$\infty$ if $eth = -1$ otherwise $0$</td>
</tr>
<tr>
<td>EnquiryFailed</td>
<td>enquirySuccessful</td>
<td>EnquirySuccessful</td>
<td>$\infty$ if $ecb = -1$ otherwise $0$</td>
</tr>
<tr>
<td>EnquirySuccessful</td>
<td>enquiryFailed</td>
<td>EnquiryFailed</td>
<td>$\infty$ if $eth = -1$ otherwise $0$</td>
</tr>
<tr>
<td>Preparing</td>
<td>prepared</td>
<td>Prepared</td>
<td>$pre$</td>
</tr>
<tr>
<td>Prepared</td>
<td>commit</td>
<td>Committing</td>
<td>$-pre$</td>
</tr>
<tr>
<td>NotPrepared</td>
<td>callback</td>
<td>PrepareCallback</td>
<td>$\infty$ if $pcb = -1$ otherwise $0$</td>
</tr>
<tr>
<td>Committing</td>
<td>committed</td>
<td>Committed</td>
<td>$com$</td>
</tr>
<tr>
<td>NotCommitted</td>
<td>callback</td>
<td>CommitCallback</td>
<td>$\infty$ if $ccb = -1$ otherwise $0$</td>
</tr>
<tr>
<td>Compensating</td>
<td>compensating</td>
<td>Failed</td>
<td>$cmp$</td>
</tr>
<tr>
<td>Successful</td>
<td>undo</td>
<td>Failed</td>
<td>$undo$</td>
</tr>
</tbody>
</table>

Table 5.1: Cost for a client to transition from one state to another.

$cmp \in \mathbb{R}$ is the cost of a successful compensate action. A negative value indicates an amount paid back to the client. A value of $\infty$ indicates that the operation is not supported.

Given this definition, Table 5.1 shows the non-zero edge weights for the client state-transition diagram, which represent the cost of the transitions for a given contract. For example, if a client sends a commit message from the Active state, requesting that the service in the interaction be committed, the client is charged $com$ if the provider successfully commits the operation and replies with a committed message. If call backs are not supported, then the edge associated with the call back is given a weight of $\infty$ to indicate that it is impossible. Of particular interest is the cost of transition when sending a commit message from the Prepared state. The interaction transitions to the Committing state and refunds the price of the previously-completed prepare operation. This stops the overall price of committing after first preparing from exceeding the cost of initially committing the action.
CHAPTER 5. A FORMAL MODEL OF THE DYNAMIC TRANSACTION SYSTEM

5.2.1.1.3 Definition Of Interaction

An interaction \((s, c, t, \delta, u) \in I = S \times C \times \mathbb{N} \times \mathbb{N} \times \mathbb{R}\) is a client interaction in which:

- \(s\) is the current state of the interaction, as described in Section 5.2.1.1.1.
- \(c\) is the contract being used for the interaction, as described in Section 5.2.1.1.2.
- \(t\) is the time that the interaction entered this state. This is important to allow client reasoning about timeouts, as will be seen in Section 5.2.2.
- \(\delta\) is the length of time before a timeout actually occurs during which the client behaves as if the timeout has occurred. Since communication between clients and providers is not instantaneous, any timeout effectively occurs before the actual time specified. \(\delta\) allows the client to model this.
- \(u\) is the cost to undo an action once it has been successfully completed. Even if a service call does not offer a compensatory action, a client may still be able to logically (to the client, at least) undo the action that has been performed. \(u\) represents the cost required to perform that action. A value of \(\infty\) indicates that such an action is not possible.

5.2.1.2 Combining Interactions Into Workflow Interactions

Section 5.2.1.1 describes a single client interaction with a provider. However, a client workflow will typically involve multiple interactions combined to be arbitrarily complex. A workflow comprises a set of actions, where each action is either an interaction with a single provider, or a workflow. Workflows can be executed in sequence or in parallel, and may provide an alternate workflow to be attempted in case the initial workflow fails. Based on \(\pi\)-calculus\(^1\) [82], this section describes a model for client workflows.

\(^1\) \(\pi\)-calculus is a process calculus that allows the formal modelling of processes executing on concurrent systems. The calculus presented here is based on \(\pi t\)-calculus [17, 16], but removes its inherent transactional support to allow the use of the more-general transactional properties examined in this work.
5.2.1.2.1 Definition Of Client Workflow Interaction

Let $e$ be a special client-defined interaction that has a contract which supports only the commit operation (which is always guaranteed to succeed), with no time outs and zero cost associated with all actions. In this case, a client workflow interaction $W$ is defined by the following syntax:

$$ W \triangleq \text{done}\ (\text{success}) \quad | \quad \text{abort}\ (\text{failure}) \quad | \quad (p)i\ (\text{client interaction}) \quad | \quad W|W\ (\text{parallel}) \quad | \quad W;W\ (\text{sequence}) \quad | \quad W, W\ (\text{alternative}) $$

$\text{done}$ and $\text{abort}$ are activities that respectively indicate the successful or unsuccessful completion of a workflow. $(p)i$, where $i \in I \cup \{e\}$ is a client interaction, either as specified in Section 5.2.1.2, or $e$ as defined above$^2$, and $p$ is a unique identifier for that interaction (in particular, each interaction with a provider can be distinguished by the value of $p$).

The parallel operation $W|W$ executes two workflows in parallel, with no guarantee as to which will complete first. In contrast, the sequence operator $W;W$ guarantees that the workflow on the right will only succeed if the workflow on the left succeeds first. Finally, the alternative operator $W,W$ means that the workflow succeeds if either the first workflow succeeds and the second fails, or, given the first workflow’s failure, the second workflow succeeds (i.e. only one of the two ever succeeds)$^3$.

---

$^2$Inclusion of $e$ allows the specification of optional components to the workflow; $V,e$ can complete successfully if $V$ fails by having $e$ succeed.

$^3$This is similar to the $+$ operation in $\pi$-calculus [82], but differs because of the two end states and the use of client interactions rather than the sending and receiving of messages.
Given the above definition, workflow interactions $V, W, X$, and interaction $i = (s, c, t, \delta, u)$, the following structural congruences hold:

\[
\begin{align*}
V | W & \equiv W | V \\
V | (W | X) & \equiv (V | W) | X \\
V ; (W ; X) & \equiv (V ; W) ; X \\
done | W & \equiv W \\
abort | abort & \equiv abort \\
done ; W & \equiv W \\
abort ; abort & \equiv abort \\
\end{align*}
\]

\[
\begin{align*}
V, (W, X) & \equiv (V, W), X \\
abort, W & \equiv W \\
V, abort & \equiv V \\
\end{align*}
\]

Along with $\alpha$-renaming [82], the above congruences equate all processes that never need be distinguished for any reason. The congruences in the first group are directly analogous to those in $\pi t$-calculus [17, 16]. The second group specifies that $\_\_\_$ is associative and that $\abort$ is an identity to the operation.

### 5.2.1.2.2 Definition Of Client Workflow

$(w, t, S, F) \in C = W \times N \times R \times R$ is a client workflow in which:

- $w$ is the client workflow interaction to be completed.
- $t$ is the time that the client entered this state.
- $S$ is the maximum amount the client is willing to spend to have the client workflow interaction end successfully.
- $F$ is the maximum amount the client is willing to spend to have the client workflow interaction end unsuccessfully.
5.2.2 Determining A Client’s Next Action

The above definition of client workflow represents the state of the workflow at a particular time. However, workflows are not static, and change over time. Thus, it is necessary for the client to determine which actions it should perform to transition into a new state. For the purpose of a client workflow, the only actions that can be performed are the sending of messages to a provider in the workflow, waiting for a message from one of the providers in the workflow, or undoing a completed operation. This section describes how a client can determine the next action to be taken.

In all cases, the client will not perform any action that could require it to exceed one of its budgets. Since each workflow is a combination of participant interactions, and each interaction starts in the Initial state, in which the client can cause the interaction to fail at no cost, it is enough to ensure that the client does not perform any action that may cause the cost of the workflow to grow larger than its budgets. In this way, the client’s correct behaviour can be guaranteed, so that the workflow will definitely end in a state that the client finds acceptable. In order to allow this reasoning, it is necessary for a client to be able to calculate costs associated with a particular workflow, and with the actions it can perform from that workflow. Section 5.2.2.1 specifies how the client can calculate the costs of a single interaction, and Section 5.2.2.2 extends these calculations to workflow interactions. These cost calculations, and the possible state transitions for individual provider interactions, allow the client to determine the set of actions it can perform from any given state, as described in Section 5.2.2.3. Once the client has determined which actions are possible, the client must choose an action to perform. Section 5.2.2.4 describes how a client can make this decision.

5.2.2.1 Calculating The Cost Of A Single Interaction

Client workflows begin with each interaction in their workflows in the Initial state, and can then send messages to the providers in the interactions to transition between the states shown in Figure 5.2. Clients can use the pricing information for each transition between states, as specified in Section 5.2.1.1.2, to reason about costs for the rest of the interaction. In particular, the client can determine the minimal cost required to have the interaction end in the Successful state, or the maximum amount it would cost to transition the interaction from its current state to the Failed state. These calculations are described,
respectively, in Section 5.2.2.1.1 and Section 5.2.2.1.2.

5.2.2.1.1 Minimum Success Cost

Table 5.2 shows the minimal cost, \( s : S \rightarrow \mathbb{R} \), required from each state of an interaction to reach the \textit{Successful} state. Once an activity has failed, it is impossible for the activity to succeed, so \( s(\text{Failed}) = \infty \). Similarly, an interaction already in the \textit{Successful} state has no cost to achieve the \textit{Successful} state, and so \( s(\text{Successful}) = 0 \). Working backwards, it is possible to calculate the minimal success cost for all other states by considering the cost to transition between states, and the cost from the new state. For example, \( s(\text{Compensating}) = \text{cmp} + s(\text{Failed}) = \infty \), since the only possible transition from the \textit{Compensating} state is to the \textit{Failed} state, at a cost of \text{cmp}. Thus, it is impossible to succeed from the compensating state. Similarly, from the \textit{Enquired} state, the interaction can transition to either the \textit{EnquirySuccessful} or the \textit{EnquiryFailed} states. In each case, the cost of that transition is \( \text{enq} \). Thus \( s(\text{Enquired}) \) is the minimum of \( \text{enq} + s(\text{EnquirySuccessful}) \) and \( \text{enq} + s(\text{EnquiryFailed}) \). It is assumed that \( \text{enq} \geq 0 \), or else the client’s best option would be to continually make enquiries, increasing the client’s budget, until the cost of the workflow was negligible.

Given interaction \( I \ni i = (s, c, t, \delta, u) \), the minimum success cost of interaction \( i \) is \( s(i) = s(s) \).

5.2.2.1.2 Maximum Failure Cost

While it is important for the client to calculate the minimal amount required to end an interaction successfully, it is equally important to be able to calculate the cost to complete the interaction unsuccessfully. This is the maximal cost, \( f \), that the client must spend to move the interaction to the \textit{Failed} state. Calculation of \( f \) is specified in Table 5.3. For any state from which the client can make a transition to the \textit{Failed} state with no possibility of the provider sending a message to alter the state, the maximal cost of failure from that state is the cost of the transition. For example, \( f(\text{Prepared}) = 0 \), since the client can choose to cancel the interaction from that state with no cost. If a message from the provider can change the current state of the interaction, then the value of \( f \) is the maximal value for all possible states into which the interaction could be transitioned. For example, \( f(\text{NotCommitted}) = \max(f(\text{Failed}), f(\text{CommitCallback})) \), since the client could
### Table 5.2: Minimum cost for client service interaction to succeed from given state.

<table>
<thead>
<tr>
<th>State</th>
<th>Minimum Success Cost (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>( \min{s(Active), s(Failed)} = \text{com} )</td>
</tr>
<tr>
<td>Active</td>
<td>( \min{s(Enquired), s(Preparing), s(Committing), s(Failed)} = \text{com} )</td>
</tr>
<tr>
<td>Enquired</td>
<td>( \min{enq + s(EnquirySuccessful), enq + s(EnquiryFailed)} = enq + \text{com} )</td>
</tr>
<tr>
<td>EnquiryFailed</td>
<td>( \min{s(Enquired), s(Preparing), s(Committing), s(Failed)} = \text{com} )</td>
</tr>
<tr>
<td>EnquirySuccessful</td>
<td>( \min{s(Enquired), s(Preparing), s(Committing), s(Failed)} = \text{com} )</td>
</tr>
<tr>
<td>Preparing</td>
<td>( \min{pre + s(Prepared), s(NotPrepared)} = \text{com} )</td>
</tr>
<tr>
<td>NotPrepared</td>
<td>( \min{s(PrepareCallback), s(Failed)} = \text{com} )</td>
</tr>
<tr>
<td>PrepareCallback</td>
<td>( \min{s(Preparing), s(Failed)} = \text{com} )</td>
</tr>
<tr>
<td>Prepared</td>
<td>( \min{-pre + s(Committing), s(Failed)} = -pre + \text{com} )</td>
</tr>
<tr>
<td>Committing</td>
<td>( \min{\text{com} + s(Committed), s(NotCommitted)} = \text{com} )</td>
</tr>
<tr>
<td>NotCommitted</td>
<td>( \min{s(CommitCallback), s(Failed)} = \text{com} )</td>
</tr>
<tr>
<td>CommitCallback</td>
<td>( \min{s(Committing), s(Failed)} = \text{com} )</td>
</tr>
<tr>
<td>Committed</td>
<td>( \min{s(Successful), s(Compensating)} = 0 )</td>
</tr>
<tr>
<td>Compensating</td>
<td>( cmp + s(Failed) = \infty )</td>
</tr>
<tr>
<td>Successful</td>
<td>0</td>
</tr>
<tr>
<td>Failed</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>
### Table 5.3: Maximum cost for client service interaction to fail from given state.

<table>
<thead>
<tr>
<th>State</th>
<th>Maximum Fail Cost ($f$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>$f(\text{Failed}) = 0$</td>
</tr>
<tr>
<td>Active</td>
<td>$f(\text{Failed}) = 0$</td>
</tr>
<tr>
<td>Enquired</td>
<td>$\max{enq + f(\text{EnquiryFailed}), enq + f(\text{EnquirySuccessful})} = enq$</td>
</tr>
<tr>
<td>EnquiryFailed</td>
<td>$f(\text{Failed}) = 0$</td>
</tr>
<tr>
<td>EnquirySuccessful</td>
<td>$f(\text{Failed}) = 0$</td>
</tr>
<tr>
<td>Preparing</td>
<td>$\max{f(\text{NotPrepared}), pre + f(\text{Prepared})} = \max{0, pre}$</td>
</tr>
<tr>
<td>NotPrepared</td>
<td>$f(\text{Failed}) = 0$</td>
</tr>
<tr>
<td>PrepareCallback</td>
<td>$f(\text{Failed}) = 0$</td>
</tr>
<tr>
<td>Prepared</td>
<td>$f(\text{Failed}) = 0$</td>
</tr>
<tr>
<td>Committing</td>
<td>$\max{\text{com} + f(\text{Committed}), f(\text{NotCommitted})} = \max{\text{com} + \min{\text{undo}, \text{cmp}}, 0}$</td>
</tr>
<tr>
<td>NotCommitted</td>
<td>$\max{f(\text{Failed}), f(\text{CommitCallback})} = 0$</td>
</tr>
<tr>
<td>CommitCallback</td>
<td>$f(\text{Failed}) = 0$</td>
</tr>
<tr>
<td>Committed</td>
<td>$\min{f(\text{Successful}), f(\text{Compensating})} = \min{\text{undo}, \text{cmp}}$</td>
</tr>
<tr>
<td>Compensating</td>
<td>$\text{cmp} + f(\text{Failed}) = \text{cmp}$</td>
</tr>
<tr>
<td>Successful</td>
<td>$\text{undo}$</td>
</tr>
<tr>
<td>Failed</td>
<td>$0$</td>
</tr>
</tbody>
</table>
transition to the *Failed* state directly, but a call back from the provider would move it to the *CommitCallback* state instead.

Given interaction $I \ni i = (s, c, t, \delta, u)$, the maximum failure cost of interaction $i$ is $f(i) = f(s)$.

### 5.2.2.2 Calculating The Costs Of A Workflow Interaction

Using the definition of minimum success cost for a single interaction described in Section 5.2.2.1.1, and the maximum fail cost in Section 5.2.2.1.2, it is possible to calculate $s$, the minimum success cost for a workflow interaction, and $f$, the maximum failure cost for a workflow interaction, as follows:

\[
\begin{align*}
    s(& \text{done}) = 0 \\
    s(& \text{abort}) = \infty \\
    s(& V|W) = s(V) + s(W) \\
    s(& V; W) = s(V) + s(W) \\
    s(& V, W) = \min\{s(V) + f(W), f(V) + s(W)\} \\
    f(& \text{done}) = \infty \\
    f(& \text{abort}) = 0 \\
    f(& V|W) = f(V) + f(W) \\
    f(& V; W) = f(V) + f(W) \\
    f(& V, W) = f(V) + f(W) \\
    s(& (p)i) = s(i) \\
    f(& (p)i) = f(i)
\end{align*}
\]

*done* represents the successful completion of a workflow interaction, and thus has a success cost of 0 and a failure cost of $\infty$ (as it is not undoable). Similarly, *abort* represents a failure of the workflow, so its failure cost is 0 and its success cost is $\infty$. Both the success and failure costs of workflows performed in parallel ($V|W$) or in sequence ($V; W$) are simply the sums of associated costs of each component included in the workflow. Alternatives such as ($V, W$) are successful whenever exactly one of the alternatives succeeds, making
the minimum success cost the minimum cost required to have one of the alternatives fail and the other succeed. The failure cost of alternatives is the sum of the failure cost of each component, as both must fail for the alternatives to fail. Finally, the success or failure cost of an interaction with a particular provider is the success or failure cost of the interaction.

5.2.2.3 Defining Allowed Actions

From any point in a workflow, a client may have many different actions it can perform. These actions will either be to send a message to a provider or wait for a reply to a message that has already been sent. The messages that can be sent or received in a workflow depend on both the state of the workflow and the costs associated with sending or receiving those messages. Through careful calculation the client can guarantee that the workflow will only ever succeed within the success budget, or fail within the fail budget.

5.2.2.3.1 Next Actions For A Single Interaction

The possible next actions for a single interaction are essentially the possible transitions in Figure 5.2 that the client can initiate, as well as the option to wait. Given interaction \( i = (s, c, t, \delta, u) \), the set of next possible actions at time \( t_1 \) is defined as \( \text{next}(i, t_1) = \text{next}(s) \) if there is no timeout within time \( \delta \) of \( t_1 \), or the timeout action if a timeout occurs within time \( \delta \) of \( t_1 \), where \( \text{next}(s) \) is defined as in Table 5.4, and the time out behaviour is as specified in the interaction’s contract.

The client can choose to perform any action in its next set. A wait action moves a workflow \( W = (w, t, S, F) \) to \( W' = (w, t + 1, S, F) \). Any other action reduces \( S \) and \( F \) based on the cost of the action, and moves the corresponding interaction in \( w \) to the state specified by the transition. For example, consider the workflow \( W = (w, t, S, F) \), in which \( w = (p)(Prepared, c, t, \delta, u) \) (where \( p \) is a unique identifier for a provider interaction). Then, given \( t_1 > t \), \( \text{next}(w, t_1) = \text{p.next}((\text{Prepared}, c, t, \delta, u), t_1) = \{\text{p.commit}, \text{p.cancel}, \text{p.wait}\} \) unless, given \( \text{pto} \) from the contract \( c \), \( t + \text{pto} > t_1 - \delta \) (in which case \( \text{next}(w) = \{\text{p.commit}\} \) if \( \text{pto} \geq 0 \), or \( \text{next}(w) = \{\text{p.cancel}\} \) if \( \text{pto} < 0 \) ). If \( t + \text{pto} < t_1 - \delta \), then \( \text{p.commit} \) would move \( W \) to \( W' = (w', t + 1, S - \text{com}, F - \text{com}) \), where \( w' = (p)(\text{Committing}, c, t_1, \delta, u) \), since \( \text{commit} \) has cost \( \text{com} \) and moves the given interaction to the Committing state.
### 5.2.2.3.2 Next Actions For A Workflow

The possible next actions of an entire workflow interaction at time $t$ can be calculated as follows:

\[
\begin{align*}
    \text{next}(\text{done}, t) &= \{} \\
    \text{next}(\text{abort}, t) &= \{} \\
    \text{next}(V|W, t) &= \text{next}(V, t) \cup \text{next}(W, t) \\
    \text{next}(V; W, t) &= \text{next}(V, t) \cup \text{next}(W, t) \\
    \text{next}((V, W), t) &= \text{next}(V, t) \cup \text{next}(W, t) \\
    \text{next}((p)i, t) &= p\text{.next}(i, t)
\end{align*}
\]

Once the client has determined the set of actions that it is possible for it to perform next, the client can reason about the cost of performing each possible action. The cost of an individual action is 0, unless otherwise specified in Table 5.1, and the client can reason about the minimum success cost and maximum fail cost once the cost of the transition is removed from all budgets. If any of the possible actions result in a workflow that exceeds one of the client’s budgets, the action is removed from the set of potential actions. After the possible actions that could exceed the client’s budgets are removed, the remaining

<table>
<thead>
<tr>
<th>State</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>{accept, reject, wait}</td>
</tr>
<tr>
<td>Active</td>
<td>{enquire, prepare, commit, cancel, wait}</td>
</tr>
<tr>
<td>Enquired</td>
<td>{wait}</td>
</tr>
<tr>
<td>EnquiryFailed</td>
<td>{enquire, prepare, commit, cancel, wait}</td>
</tr>
<tr>
<td>EnquirySuccessful</td>
<td>{enquire, prepare, commit, cancel, wait}</td>
</tr>
<tr>
<td>Preparing</td>
<td>{wait}</td>
</tr>
<tr>
<td>NotPrepared</td>
<td>{cancel, wait}</td>
</tr>
<tr>
<td>PrepareCallback</td>
<td>{prepare, commit, cancel, wait}</td>
</tr>
<tr>
<td>Prepared</td>
<td>{commit, cancel, wait}</td>
</tr>
<tr>
<td>Committing</td>
<td>{commit, cancel, wait}</td>
</tr>
<tr>
<td>NotCommitted</td>
<td>{notCompleted, wait}</td>
</tr>
<tr>
<td>CommitCallback</td>
<td>{commit, cancel, wait}</td>
</tr>
<tr>
<td>Committed</td>
<td>{close, compensate, wait}</td>
</tr>
<tr>
<td>Compensating</td>
<td>{wait}</td>
</tr>
<tr>
<td>Successful</td>
<td>{undo, wait}</td>
</tr>
<tr>
<td>Failed</td>
<td>{}</td>
</tr>
</tbody>
</table>

Table 5.4: Set of next possible actions that can be performed in a single interaction.
actions form the set of valid actions that the client can perform.

5.2.2.4 Choosing The Next Action

Once the client has a set of valid actions that can be made at a given time, the client must choose to perform one of those actions. The action to perform could be chosen from the valid set completely at random and the result would be a legal workflow interaction, but the client may choose a different strategy based upon its needs. For example, the client may choose the action that minimises the total cost for the client but still ends the workflow successfully. Alternatively, the client may choose to minimise the cost in case of workflow failure, or minimise some other measurable property of the workflow. For example, if the client wishes to reduce the uncertainty of the success of its workflow, it may choose *enquire* actions to see if the providers being utilised in the workflow have the required resources before any single interaction is sent a *commit* request. Regardless of the strategy used, the client performs the action that it chooses. The performance of the action moves the workflow into a new state (which may be nothing or at times a step if the client has chosen to wait), and the client continues the process of choosing an action to be performed until the workflow is completed.

Initially, the aim of the client is to complete the workflow successfully. If that becomes impossible then the aim shifts to attempting to complete the transaction. In order to achieve this aim, the client can use the structural congruence laws described in Section 5.2.1.2.1 to simplify the workflow as much as possible, and then use the reduction rules defined below:

- Any client interaction in the *Failed* state is replaced with abort. This indicates that a client interaction in the *Failed* state is an unsuccessful action.

- Any workflow that has no alternatives and all client interactions in the *Successful* state is replaced with done. This ensures that only one of any alternatives complete successfully, as alternatives are removed through the structural congruence rules only in the case of abort. Further, it states that any sequence or set of workflows running in parallel succeed if each workflow in the sequence or set complete successfully.

Thus, a client’s first aim should be to move individual interactions into the *Successful* state, unless there is another alternative for the interaction (or set of interactions) that
is already in the *Successful* state. When there are alternatives in a workflow, the client
only wants one of the alternatives to have all of its interactions in the *Successful* state;
all other alternatives should have their interactions moved to the *Failed* state, which the
structural congruence rules can then remove from the workflow. If, however, the client
has an interaction that does not have an alternative move into the *Failed* state, then it
will not be possible to complete the workflow successfully, and all individual interactions
must be transitioned into the *Failed* state.

It should be noted that, since the client will only ever choose an action that leads to
an acceptable state (either ending in a *Successful* state after having spent no more than
the success budget, or in the *Failed* state after having spent no more than the fail budget),
the presented system ensures that any failure of a particular service is handled correctly.
In particular, the client will only request a service if it is acceptable to have that service
fail. Thus, the failure of a service, or even the failure of a transaction, is not seen as an
error, but as a valid state modelled by the system. Any cascading effects caused by a
service failure are similarly included in the model, ensuring that the client always ends in
a deterministic state.

### 5.2.3 Reasoning With Incomplete Information

To this point, it has been assumed that the client has a contract offer for each interaction
it wishes to use in the client workflow. However, in some cases, such as when the client is
anxious to complete the workflow as quickly as possible, the client will wish to begin its
workflow before all of this information is available. Starting a workflow with incomplete
information is inherently risky; the client will not know the minimum success cost of the
interactions in its workflow, and will thus be unable to calculate the minimum success cost
for the workflow. Note, however, that the client will be able to determine the maximum
failure cost, as a workflow without a contract can be cancelled at zero cost (since the
transition of an interaction from *Initial* to *Failed* has no cost).

One way that a client can handle a workflow with incomplete information is to estimate
the minimum success cost of any interaction in the workflow for which it does not yet
have a contract, and add the possibility of waiting for a concrete contract rather than
immediately cancelling any workflow that exceeds the success budget. On receipt of an
actual contract, the client replaces its estimate with the calculated cost, and only cancels
the workflow based on the success budget if the definite minimum success cost is too high. If the client is extremely risk adverse, then it can estimate the minimum success cost to be $\infty$ for all actions, guaranteeing that it will wait until all required contracts have been received before continuing processing of the workflow. A client willing to take more risk will estimate a lesser cost, and may thus begin processing the workflow while still waiting for further contracts. Even when estimating a lesser cost, the client is guaranteed to end in an acceptable state; the maximum fail cost is guaranteed not to exceed the fail budget, so the client will always be able to end the workflow unsuccessfully but within the fail budget. However, the client may begin a workflow that is impossible to complete successfully, and will have thus needlessly paid for some operations when the entire workflow never had a chance of success.

The estimation process also provides the client with a way to prioritise alternative workflows. Rather than requesting contracts for all alternative workflows, the client may only request the contracts for the first alternative, and estimate that all later alternatives would have a higher cost. In this way, the client would only attempt the first alternative. If the first alternative fails, however, the client may adjust its estimate and request any required contracts for the next alternative. Thus all alternatives would be tried in order, and the workflow would only fail if all alternatives failed.

Using the techniques in Section 5.2.2 and the client’s estimates of the minimum success cost of any action for which the client has not yet been offered a contract, the client is automatically able to determine the actions to take to process its workflow and stay within budget. This guarantees that the client ends in an acceptable state and that its actions are correct according to the dynamic transaction protocol. Thus, as long as providers also follow the protocol correctly, all participants end in a valid state, which shows the correctness of the protocol.

5.3 Negotiating Transaction Levels

So far, this chapter has assumed that clients and providers have already agreed on the level of transaction support that will be used for a client-provider interaction. However, as mentioned in the previous chapter, it is possible for clients and providers to negotiate the level of transaction support that will be offered for a particular interaction (see Section
4.3.2.1 for the elements of a contract that are negotiable). It can be useful to think of these negotiations in isolation, between one client and one provider, for one interaction. The negotiation can then be considered as a one-to-one negotiation using an *alternating offers* bargaining model [120]. However, providers may have many clients competing for the same resources, and the interaction is only a part of the client’s overall transaction. This means that the negotiations really are many-to-many, with any single negotiation between a provider and a client potentially depending on the actions of parties unknown to either the client or the provider.

Even when considering interactions as one-to-one between an individual client and an individual provider, the negotiation scenario is quite complex. This is because it is a *multiple-issue* scenario [120], with details hidden from both sides of the negotiation. The client does not know whether the resources offered by the provider are plentiful and easily available, or whether they are highly contended. The provider does not know whether its cooperation is necessary for the client’s overall transaction, the budget the client has available, or the level of risk the client is willing to accept. Thus neither party can be certain whether a particular offer will be seen as a concession from the other party. For example, if a client is very risk adverse, but has a plentiful budget, then having a provider lower the price for a contract offering only tentative holds may not be any more attractive than the same contract with a higher price.

This section describes some simple negotiation strategies that can be used by service providers and clients in the dynamic transaction system. Section 5.3.1 presents a strategy that can be used by a provider when negotiating with a single client, and Section 5.3.2 describes a strategy a client can use throughout its entire transaction. More complex analysis of negotiation techniques is beyond the scope of this thesis.

### 5.3.1 A Provider Negotiation Strategy

Section 5.1.2 described how providers can determine the levels of transaction support the provider is able to support for a particular interaction. One of the simplest negotiation strategies is thus to offer all possible levels available and have the client choose the one it prefers. It is still up to the provider, however, to determine the price of each of these offers. This strategy assumes that the provider has a fixed length for all timeouts, and predefined functions to calculate the cost of each of the operations supported by the
Let \(enq(r, R), ecb(r, R), eth(r, R), pre(r, R), pcb(r, R), com(r, R), ccb(r, R), cmp(r, R)\) be the functions to calculate the cost for a client to have (respectively) an enquire action, a callback after a failed enquiry, a callback on the loss of a tentative hold, a prepare action, a callback after a failed prepare, a commit action, a callback after a failed commit, or a compensate action for \(r\) resources when the provider has \(R\) available (i.e. in the Free state). A cost of \(\infty\) for any of these functions indicates that the action is not supported by the provider. Once the provider has determined the levels of transaction support that are possible, it creates a contract for each level using these functions. This set of contracts is then sent to the client as non-negotiable templates, and the client can choose to either accept one contract from the set, or to reject all of the contracts and not have any further interaction with the provider.

### 5.3.2 Client Negotiation Strategies

Section 5.2.3 described how a client can begin a transaction when it does not yet have all contracts required for the workflow that it is attempting. A client can use a similar strategy to attempt to negotiate for a set of contracts that allow the client to successfully complete its workflow. The client can begin by estimating the cost of each interaction it requires for its workflow, and then request contract templates for those interactions. If an offered contract has a cost that is no worse than the estimate made by the client, the client can accept that contract immediately. If however, the contract exceeds the estimated cost, the client may choose to negotiate with the provider to bring the cost down to within the estimated range. If such a negotiation succeeds, the client can accept the new contract. If the negotiation fails, and there is no alternative in the workflow for the interaction, the client can wait to see if the other interactions in the workflow cost less than their estimates to allow the client to increase the price that it is willing to pay for the offending interaction.

### 5.4 Summary

This chapter described a formal model of the dynamic transaction system introduced in Chapter 4. This model will be validated in future chapters through the use of a
simulator that demonstrates the model’s effectiveness. The model comprises of two parts; the provider-side, and the client-side. The provider-side component tracks the state of a provider’s resources to ensure that the provider only offers guarantees that it can definitely support. This ensures correct operation of the provider while still allowing the provider significant freedom as to the level of transaction support it will offer for a client request.

The client-side component of the model describes workflows using a syntax based on $\pi$-calculus, and a notion of client budgets to determine the valid actions that a client can perform at any time. The client can use this set of actions and a client-specified strategy to choose an action to be performed to continue the workflow. Using structural congruence and reduction rules, the client can calculate various costs for the remainder of its workflow and ensure that these fall within predefined client budgets. By ensuring that the maximum cost to cancel the workflow is always within the client’s failure budget, the client is guaranteed to end in a state that it finds acceptable. Further, by considering the minimum success budget, the client is able to cancel any workflow that is impossible to complete successfully within budget as soon as it is detected that the workflow would be too expensive to complete.

The model considered in this chapter assumes that clients have already received acceptable contract templates for all interactions required for the workflow. If this is not the case, it is possible for providers and clients to negotiate both the offered level of transaction support, and the cost for each component of the contract. While complete consideration of such negotiation strategies is out of the scope of this work, this chapter did introduce a strategy for providers and a strategy for clients that can be used when negotiating transaction templates.

The following chapters use the model presented in this chapter as the basis of a simulator for the dynamic transaction system. The simulator allows the practical aspects of the model to be tested, and allows the comparison of different transaction strategies for both service providers and clients.
Chapter 6

Simulating The Dynamic Transaction System

Let’s take flight simulation as an example. If you’re trying to train a pilot, you can simulate almost the whole course.

Roy Romer

Web Services transactions have been shown to be very different from traditional transactions. Multiple parties work together to achieve a client’s aims, but still wish to remain autonomous. Further, the different service providers may have different levels of transactional support, making verification and testing of different transaction schemes very complex. While theoretical analysis is essential to ensure that transaction schemes work correctly, these models (e.g. [29, 45]) do not easily allow comparison between different transaction techniques. In particular, these models make it difficult to determine which transaction schemes are best suited for certain conditions, applications, or environments.

One option to test the practical performance of Web Services transactions techniques is to set up a real system and perform experiments on that set up. Since Web Services transactions often cross boundaries between organisations and utilise services offered by providers that are geographically distant, any such environment used to test real-world performance would be required to be quite large. Such a set up is costly, and any results may be dependent on uncontrollable variables that have no relation to what is being tested. For example, if one transaction scheme was tested while a large amount of unrelated network traffic was being sent, any results may unfairly be considered worse than another
scheme that was tested when external network traffic was low. If, rather than a large network, a smaller, local network was used to compare the transaction schemes, these problems could be monitored or avoided. However, the results would only be applicable for that local network, and would not necessarily reflect real-world performance.

Instead, simulation can be used to provide an indication of practical results. In simulation, some or all of a system is abstracted so that only the features important to the current investigation are tested [66]. When simulating Web Services transactions, details such as the networking topology, the timing of events, and the actual services being used can be abstracted. This can allow intricate comparison of various transaction strategies, by allowing the parameters of interest to be studied while ensuring that all other factors are kept constant.

Most available Web Services simulation environments replace a Web Service with a simple, usually local, program that sends and responds to messages in a way that is appropriate to the service being simulated. However, when examining Web Services transactions, further abstraction can occur [100]. Messages do not have to be sent in the exact same format as with real services; it is only the transaction interaction patterns that need to be simulated. Accordingly, the Web Services transaction simulator presented in this chapter models transaction flow rather than service flow.

The remainder of this chapter is organised as follows. Section 6.1 details the general operation of the Web Services transactions simulator, and Section 6.2 details how participants are modelled in a simulation. Section 6.3 describes the input to the simulator and how it can be used to enter different scenarios and transaction techniques into the simulator. Section 6.4 then describes the output of the simulator, and how it can be interpreted to allow a fair comparison of the transaction schemes being used. Finally, Section 6.5 concludes the chapter with a summary of the simulator’s operation.

6.1 The Web Services Transactions Simulator

Web Services are based on the exchange of messages between a sender and a receiver. On receipt of a message, the receiver processes the message, which may alter the receiver’s state, and may send further messages as a result, possibly to third parties. The Web Services transactions simulator follows this messaging model. Each participant in a
CHAPTER 6. SIMULATING THE DYNAMIC TRANSACTION SYSTEM

Algorithm 6.1 The main loop for the transaction simulator

set current time to 0
while (message queue is not empty or there exists a participant that has not completed) {
    while (first message in message queue has received time before or equal to current time) {
        Remove first message from message queue
        and pass to correct recipient
    }
    for (each participant in the simulation) {
        for (each message that the participant has to send at the current time) {
            Calculate received time for message
            Place message in message queue, prioritised by received time
        }
    }
    increment current time
}

scenario being simulated is modelled as an object that communicates by passing messages through the central simulator. The main algorithm used by the simulator, which has been implemented in Java, is specified in Algorithm 6.1.

The simulator continually executes a loop until all participants report that they have completed their operations, and all messages have been sent. Each iteration of the loop increases the current time in the scenario by one. The simulator maintains a queue of messages to be delivered and, at each time step, consults the queue to determine which messages should be delivered at the current simulation time. The messages found during this process are delivered to the intended recipients for processing. Once the simulator has delivered all messages required at the current timestep, the simulator queries each participant to receive any messages that the participant wishes to send at that time. Since message transmission is not instantaneous, the simulator then calculates when the messages will be received before adding them to the simulator’s message queue in the appropriate order.

Section 6.1.1 provides details of how messages are modelled in the simulator. It is important to note that, since messages are being sent between participants, each message takes time to travel from the sender to the receiver. Further, the receiver requires time to process the message. As each participant is autonomous, each may process messages
at a different speed, and the time it takes for a message to travel from one participant to another will depend on the network connection of both parties. Thus, it is important that the simulator is able to model complex timing information. The simulator achieves this using time modellers, as described in Section 6.1.2.

The simulator must also model the various participants in a scenario. A participant is either a concrete service provider, an abstract service provider, or a client. Service providers offer resources, and clients consume resources. Section 6.2 describes how participants are modelled in the simulator. Participant behaviour follows the formal model introduced in Chapter 5.

6.1.1 Simulating Messages

The Web Services transaction simulator is responsible for passing messages between the different participants in accordance with the scenarios being modelled. When a participant sends a message, it knows the identity of the sender and the receiver, the content of the message, and the time the message is sent. Each message is also given an id to allow unique identification. The sender then passes the message to the simulator. The simulator calculates the time the message is to be received and ensures that the message is delivered to the receiver at that time.

Since each participant may have numerous different interactions, any time a new request is made, the message is allocated a new interaction id. Any replies or other messages that relate to that request are then given the same interaction id. This allows different interactions to be correctly and easily identified by the simulator, ensuring that the message is handled correctly.

As mentioned previously, the sender of a message does not know the time when the message will be received. This is handled by the simulator by having participants pass SentMessage objects to the simulator, which converts them to ReceivedMessage objects to be given to the intended recipient. The ReceivedMessage object specifies the time the message is received by the recipient system, as well as all other details in a SentMessage. However, before the recipient can actually examine the message, some processing must be done to allow the message to progress to the application handling the message. Thus, the simulator converts the ReceivedMessage to a ProcessedMessage, which takes this extra processing time into account, before passing it to the recipient for processing.
6.1.2 Simulating Timing

When simulating a Web Services transactions scenario, there are two different timing details that must be calculated. The first simulates network conditions, giving the time that it takes a message to be sent from one participant to another. The other important timing information is how long it takes an individual participant to process a message. Both of these timing calculations can be quite complex. In real networks, the speed at which messages can be sent is rarely constant; details such as congestion, distance, and even weather can affect a network’s speed. Similarly, in individual systems, CPU and memory load, the complexity of a message to be processed, and the priorities of the software processing the message mean that message processing time is also variable. Time modellers are used by the Web Services transactions simulator to allow such complexities to be modelled.

On receipt of a `SentMessage`, the simulator uses a `NetworkTimeModeller` to calculate the time the message will be received. The returned time is used to convert the `SentMessage` into a `ReceivedMessage`, which is passed to the desired recipient. A different `NetworkTimeModeller` can be specified for each pair of participants in the scenario being simulated. If, for a given message, there is no specified `NetworkTimeModeller` for the two communicating parties, a simulation-wide default `NetworkTimeModeller` is used for the purposes of the calculation. Using different `NetworkTimeModellers`, it is possible to model situations such as when two participants are on the same local network which is connected via a slow link to all other systems. In this case, a `NetworkTimeModeller` that specifies fast send times could be set for messages between those two participants, while all other communications use a `NetworkTimeModeller` that returns much slower send times.

When passed a `ReceivedMessage`, a participant uses a `ProcessTimeModeller` to convert the `ReceivedMessage` into a `ProcessedMessage`. The `ProcessedMessage` takes into account the time it takes after receipt of a message before the participant actually starts processing it. Each participant has a default `ProcessTimeModeller`, though each can also have different `ProcessTimeModellers` for messages from different participants. Combined with the `NetworkTimeModellers`, this allows complex specifications of timing information, such as the prioritised processing of messages from certain participants.

The main time modellers used in this thesis are random time modellers that are given
a minimum and maximum time, and calculate a random amount of time within that range whenever requested to calculate the transmission or processing time of a message. However, the NetworkTimeModeller and ProcessTimeModeller provide simple interfaces to allow extension through the use of different timing estimates, if required. As will be seen in Section 6.3, the timing information to use for each participant forms part of the input of each run of the simulator, allowing arbitrarily complex timing models to be applied for a simulation.

6.2 Simulating Participants

Participants in the simulator are entities that send and receive messages. Each participant can be involved in numerous different interactions with other participants in the simulation. At each time step, participants are queried for any messages they wish to send at that time, which are then added to the queue of messages to be sent by the simulator. The simulator passes messages from its message queue to the participants identified as the receivers of the message. On receipt of a message, a participant processes the message, as required by its current state.

Participants can be categorised as being either service providers, described in Section 6.2.1, or clients, described in Section 6.2.2. Providers offer resources that can be booked by clients. Clients typically wish to combine multiple services into a workflow. To complete their workflows, clients will typically contact numerous providers, each of which may be simultaneously communicating with many other clients.

6.2.1 Simulating Providers

A Provider is a participant that has a collection of Resources available to be booked. Each Resource has a name, and the number of resources available in the Free, Locked, Booked, and Finalised states. Clients initially contact a provider by sending a ResourceRequest, which specifies the name and quantity of a resource to be booked. The Provider then uses its ContractTemplateCreator to create a set of ContractTemplates that the Provider is willing to support for the client’s request. Each ContractTemplate specifies details such as the cost of the five basic transaction operations introduced in Chapter 4 (enquire, prepare, commit, compensate, and callback), and the lengths of time
and default operations to support for timeouts. The *ContractTemplateCreator* is an interface that can be implemented to define different provider strategies to determine the level of transaction support to offer for a request.

After replying with the set of *ContractTemplates* that the provider is willing to offer, the interaction waits until a message is sent indicating that one of the templates has been accepted, that a template should be used as the basis for renegotiation, or that all templates have been rejected. In the case where all templates are rejected, the interaction ends with no further messages being sent. If a template is sent back for renegotiation, the template is passed to the *ContractTemplateCreator*, which uses the template to create a new set of *ContractTemplates* to counter-offer. In the case that a template has been accepted, a *Contract* is created, and the interaction follows the model described in Chapter 5, guided by the details in the accepted contract.

The *Provider* continues, moving resources between the Free, Locked, Booked, and Finalised states as required by any contracts it has already accepted, and uses the *ContractTemplateCreator* to determine levels of support to offer for any new requests that it receives. If ever any of the interactions that the *Provider* is a part of are not in a final state, the *Provider* indicates to the *Simulator* that it has not yet completed. This ensures that the simulation will not end with a *Provider* part-way through an interaction.

### 6.2.1.1 Abstract Providers

The simulator also has rudimentary support for providers that offer abstract services [105]. An *AbstractProvider* is a participant that has a list of providers (possibly abstract providers), and can be used as a single point of contact for all providers in that list. When a client sends a request to the abstract provider, it passes the request to each of the providers and combines their offers into a single offer reply to the requesting client. The client then continues its workflow passing all messages through the abstract provider. The simulator allows for more complex abstract providers, such as offering services that combine services from multiple providers to give better transactional guarantees or cheaper cost, but such enhancements have not currently been implemented.
6.2.2 Simulating Clients

In contrast to Providers, who offer resources, Clients are participants that consume resources. Clients consist of a ClientWorkflow that the Client wishes to complete, a success and failure budget, which give the maximum amount the Client is willing to spend for the workflow to complete either successfully or unsuccessfully (respectively), and an ActionSorter which, when given a set of all possible actions that the Client can perform at a particular time, orders the actions to determine which action the Client will perform next. A ClientWorkflow can be either a ClientInteraction, a ParallelWorkflow, a SequentialWorkflow, or an AlternateWorkflow.

A ClientInteraction is an interaction with a single Provider. The Client initially sends a ResourceRequest, consisting of the name and quantity of the required Resource, and receives a set of ContractTemplates that the Provider is willing to offer for the request. The ClientInteraction then uses a ContractChooser to determine if any of the offered templates are acceptable. If a template is acceptable, the Client creates a Contract based on the best template chosen by the ContractChooser and informs the Provider that it accepts the Contract. If none of the templates are acceptable, the ContractChooser either offers a template to use for renegotiation with the Provider, or the Client cancels the interaction. Once a Contract has been accepted, the ClientInteraction follows the transactional pattern specified in the Contract, as described in Chapter 5; at each stage the Client can determine which actions are possible given the Client’s budgets and the current state of the interaction. The ClientInteraction completes successfully if the resources are successfully booked, or completes unsuccessfully if the interaction completes without the resources being booked.

A ParallelWorkflow consists of a set of ClientWorkflows, and only completes successfully if all of the included workflows complete successfully. At each stage, the Client examines each possible action for each of the ClientWorkflows included in the ParallelWorkflow, and considers if performance of that action would allow all workflows in the ParallelWorkflow to complete within budget. This is achieved by examining the maximum cost of attempting the action, and the cost to complete the workflows once that action has been requested. This includes actions that allow successful completion of all the workflows within the success budget of the Client, and also actions that allow the failure of all workflows within the Clients failure budget.
A *SequentialWorkflow* is similar to a *ParallelWorkflow*, but the order in which workflows complete successfully is specified. In particular, a later workflow will only complete successfully if all earlier workflows in the sequence have completed successfully. To achieve this, the *SequentialWorkflow* removes any actions that could move a workflow in the sequence to a successful state from the set of possible actions, unless that workflow is the first workflow in the sequence that is not yet in a successful state. This imposes a particular order on the completion of components of the workflow.

Finally, an *AlternateWorkflow* consists of a set of *ClientWorkflows*, and only completes successfully if exactly one of the workflows in that set complete successfully. When considering the possible actions of any particular stage of the workflow, the *Client* examines the cost of performing each possible action for each of the alternate workflows, and adds the cost of completing all of the other alternate workflows. It achieves this by considering the cost required to move all of the alternate workflows to the *Failed* state after completion of the action, and also considering the cost of moving each alternative workflow to the *Successful* state while ensuring that every other workflow moves to the *Failed* state. The set of possible actions for any stage of the *AlternateWorkflow* is thus any action possible for each alternative workflow that allows at most one of the alternate workflows to complete successfully.

At every stage of the *ClientWorkflow*, the *Client* can determine a set of possible actions that guarantee the workflow will complete within budget. However, each *Client* may have a different strategy to determine which actions to perform. To allow for this, each *Client* has an *ActionSorter*, which sorts the possible actions for each stage of the workflow. Once the possible actions are sorted, the *Client* can perform the first action in the sorted list. Unless that action is a *wait* action, which specifies that the *Client* should not perform any action at the current time, performing the action moves the *ClientWorkflow* to a new state which has a new set of possible actions. Thus the *Client* continues the process of determining possible actions from its workflow, sorting those actions, and performing the first action in the sorted list until the action to be performed is a *wait* action. The *Client* then stops performing any actions until the next time step, when the process begins again.

Thus, *Clients* process their workflows by performing actions from the set of possible actions provided by the *ClientWorkflow* at each time step until the *Client’s ActionSorter* determines that the best action for the *Client* is to wait. This allows the *Client’s workflow*
to be processed in such a way that it is guaranteed that the Client’s budgets are not exceeded. Further, the ActionSorter allows prioritising of particular actions to allow the Client complete control of its risk-taking behaviour. For example, one ActionSorter may prefer an action that minimises the cost to complete the ClientWorkflow successfully, while another may prefer commit actions even if they lead to higher costs overall. For example, if two ClientInteractions in an AlternateWorkflow are committed, then one of the completed interactions will have to be compensated or undone, which would increase cost, or the AlternateWorkflow will not be in a completed state, because more than one of the alternative workflows will be in a successful state.

6.3 Input For The Simulator

The simulator uses the Spring Framework [108] as an Inversion of Control container to allow scenarios to be specified using XML files. These files describe the various NetworkTimeModellers and Participants to be included in a simulation, and the framework’s Dependency Injection [47] converts the XML description into an actual Simulator instance at run-time. The simulation is then run using the newly-created instance, which processes messages accordingly before producing output as described in Section 6.4.

One particularly useful feature of the Spring Framework is the ability to use multiple files to specify a single simulator instance. This feature allows the isolation of transaction-related properties of Participants in a scenario description. Thus, details of all ContractTemplateChoosers and ActionSorters can be placed in one file, while details such as timing information, resources offered by providers, and client budgets and workflows are included in a different file. The simulator can then be tested by altering the values in the first file, which specifies the transactional-behaviour or participants, while the second file remains constant throughout the tests. With careful definition of the contents of the second file\(^1\), this ensures that any difference in results can be directly attributed to the change of transactional properties.

---

\(^1\)Time modellers, for example, must ensure identical results regardless of the transactional properties being used. It must be guaranteed that a message sent from participant \(p_1\) to participant \(p_2\) at time \(t\) is received by \(p_2\) at the same time regardless of any other changes to the simulator input, or any change in results may be attributable to the fluctuations in timing. The time modellers used in this thesis provide such a guarantee.
6.4 Interpreting The Simulator’s Output

In order for a simulator to be useful, it must be possible to extract meaningful information from the results. As the Web Services transaction simulator is based on a message-passing model, the output chosen is a log of all messages received. This provides a persistent record of the behaviour of the simulator, allowing detailed analysis after the simulator has finished executing. This analysis allows different runs of the simulator to be compared by extracting important features of each simulation run, such as the number of clients that complete their workflows successfully, the resource utility of providers, the average length of a workflow, or the waiting time for various participants.

The simulator outputs the messages in the following format:

<id>,<intId>: <sender> (<sentTime>) -> <receiver> (<receivedTime>): <body>

Where:

- `<id>` is the unique id for the message
- `<intId>` is the interaction id for the message
- `<sender>` is the unique name of the sender of the message
- `<sentTime>` is the time the message was sent by the sender
- `<receiver>` is the unique name of the receiver of the message
- `<receivedTime>` is the time the message was received by the receiver
- `<body>` is the body of the message

Information about the end state of each participant in the simulation is appended to the end of the message log. For clients, this includes information specifying whether the workflow was completed successfully or unsuccessfully, and the cost required to achieve that state. For providers, this includes details of the resources that the provider has remaining at the end of the simulation.

6.5 Summary

This chapter describes a simulator for Web Services transactions. By modelling transaction flow rather than service flow, the simulator is able to abstract over details
that are unimportant when studying different transaction techniques, providing an indication of practical results without the high expense that real-world testing would require. The simulator follows the messaging model common in service-oriented environments, using a simple looping structure to pass messages from sender to receiver at the appropriate time. Since timing information is very important for transaction schemes, time is handled by the simulator through the use of flexible time modellers that can simulate realistic complex conditions.

The simulator models providers as participants that offer resources and clients as participants that consume resources. Each provider has a strategy to determine the levels of transaction support it is willing to offer for any request for resources, and each client has a risk-taking behaviour to determine whether or not it will accept any such level. Clients contact the providers with which they wish to interact and enter a negotiation stage. In this stage the level of transaction support is decided and any further interaction between the provider and the client follows the appropriate transaction scheme.

XML files are used to describe scenarios for the simulator, which uses dependency injection to convert the description of a scenario to the run-time objects necessary to model the scenario. The simulator logs all messages sent, which allows important details to be extracted from the output. These details include whether clients were successful and how much they spent; how many of the provider’s resources were used and how much they were paid; and the length of transactions. Further, the log of messages offers a persistent record that allows future analysis after the simulation has completed. Chapter 7 uses this information to validate the simulator and the viability of the dynamic transaction scheme.
Chapter 7

Validation

The logic of validation allows us to move between the two limits of dogmatism and skepticism.

Paul Ricoeur

This chapter demonstrates that the simulator described in Chapter 6 accurately models Web Services transactions, and that the dynamic transaction system introduced in Chapter 4 provides benefits over current transaction standards. To this end, the chapter describes the results of two validation scenarios that have been processed by the simulator. The first of these, presented in Section 7.1, demonstrates how traditional transaction support is not always adequate for long-running transactions in service-oriented environments. The second, presented in Section 7.2, shows that the dynamic transaction scheme introduced in Chapter 4 is viable, and that it has benefits not only for individual providers or clients, but for the overall performance of all parties involved in the system.

7.1 The Need For Non-Traditional Transaction Schemes

The purpose of the scenario described in this section is to demonstrate the need for different transactional properties than those used in traditional systems, where strict isolation limits the concurrent execution of different transactions. It seems intuitively obvious that overall performance of the system will be significantly reduced when later clients are unable to continue their transactions until an earlier client has finalised the decision on whether to use the resources that it has locked. This scenario verifies that this intuition is correct, showing that lesser transactional guarantees can significantly improve (lessen) the average running time of transactions in service-oriented systems.
7.1.1 Scenario Description

This scenario, based on the experiment described in [100], tests a single provider offering 1000 resources with four possible levels of transaction support. The four levels of transactional support tested are:

- The traditional prepare/commit scheme.
- The traditional prepare/commit scheme with a time out of 500, so that any call to prepare automatically commits if not told to cancel within 500 time units of receipt of the prepare call.
- A tentative hold enquire/commit scheme.
- A semantic atomicity commit/compensate scheme.

Clients in this scenario wish to book resources from this simple provider, and also to complete some other actions. For the purpose of this scenario, there are 1000 clients who each wish to book between 1 and 10 (randomly chosen) resources from the provider. The other actions are modelled as another service call that takes between 1 and 50 time units to complete, and has a 20% likelihood of failure. In reality, this other activity could be multiple activities, but it is sufficient to model them as one grouped activity for this scenario, as the main point of interest is how the different levels of transaction support offered by the competing providers impact the system, rather than the realistic modelling of the rest of the transaction.

To model this scenario in the simulator, four different inputs are tested, each with the provider offering one of the above-mentioned transaction support levels. For each input, the cost of prepare (when supported) and commit provider actions is 1\(^1\), all other provider actions 0, and the cost for a client to undo a successfully-completed workflow interaction is 0. Each client’s success budget is set to 2, and its fail budget to 1. These costs and budgets ensure that each client can afford to have either both activities succeed, both activities fail, or one activity succeed and the other fail. The prepare cost of 1 ensures that clients do not attempt the other activity unless a prepare has been granted from the provider. In the case that one activity fails, the client will fail with a penalty whenever the activity was committed and then undone on the client side.

\(^1\)Note that the cost to prepare is refunded if ever a commit is performed, giving the service an overall cost of 1 after a successful prepare, regardless of whether the interaction fails or completes successfully.
CHAPTER 7. VALIDATION

<table>
<thead>
<tr>
<th>Transaction support</th>
<th>Provider utility (%)</th>
<th>Client successful (%)</th>
<th>Client unsuccessful without penalty (%)</th>
<th>Client unsuccessful with penalty (%)</th>
<th>Average duration of a transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>100</td>
<td>23.0</td>
<td>77.0</td>
<td>0.0</td>
<td>1316</td>
</tr>
<tr>
<td>Traditional with time out</td>
<td>100</td>
<td>24.5</td>
<td>72.8</td>
<td>2.7</td>
<td>1245</td>
</tr>
<tr>
<td>Tentative hold</td>
<td>100</td>
<td>15.6</td>
<td>57.5</td>
<td>26.9</td>
<td>228</td>
</tr>
<tr>
<td>Semantic atomicity</td>
<td>86.2</td>
<td>16.7</td>
<td>83.3</td>
<td>0.0</td>
<td>134</td>
</tr>
</tbody>
</table>

Table 7.1: Results for Scenario 1

7.1.2 Results

The results for this scenario can be seen in Table 7.1. The first column indicates the level of transaction support being tested. The “provider utility” column gives the percentage of those resources originally offered by the provider that were booked once all transactions had completed. The “client successful” column indicates how many transactions completed all operations successfully. This is in contrast to “client unsuccessful without penalty”, which shows how many transactions failed without completing any of the required operations, and “client unsuccessful with penalty”, which shows the number of transactions that end partially completed, with either the resource booking completing successfully but the rest of the transaction failing, or the resource booking failing and rest of the transaction completing successfully. The final column indicates the average duration of the client transactions for each of the different levels of transaction support.

In each case, except when semantic atomicity was offered, all of the provider’s resources were utilised. As the number of client transactions was so large and the number of resources offered by the provider so small, this is hardly surprising; many more resources would be required to satisfy all clients’ requests, so there were many clients all wishing to access the same resources. When the traditional schemes were used, the resources quickly became locked by the first clients requesting them, and all other clients were required to wait until a final decision (or time out) occurred before determining whether they had access to the resources they required. Thus, if ever a client decided against using the resources it had locked, there were numerous other clients waiting to
place a lock on the resources freed by that client. Similarly, when the tentative hold scheme was used, numerous clients had simultaneous holds on the resources. Since those holds were only invalidated when resources were actually booked, transactions only failed if the other actions in the transaction failed, or the requested resources truly were not available. In contrast, when semantic atomicity was offered, it was possible for a transaction to fail because another client’s transaction had already booked those resources. If the other transaction later compensated that booking, however, then the first transaction may have been able to succeed, but, since the resources were not available when that transaction needed them, the transaction had already failed.

Examining the results of the clients’ transactions, more transactions succeeded when a traditional scheme was used. This was because clients waited until they were guaranteed that their request for resources would be successful before attempting the rest of the transaction. This essentially reduced the number of concurrently running transactions to only those that had access to the resources they required; all other running transactions were waiting to be given access to the resources before they continued with the rest of their transaction. When a time out was included on the prepare stage, it was possible that a client may have had the required resources locked and then attempted to complete the rest of the transaction, but had the time out occur before they could complete their booking. This resulted in a failure with penalty, though the longer the time out period, the less likely this would occur. On the other hand, when tentative hold was used, many transactions were granted holds on the same resources and then attempted to complete the rest of their transaction. Thus it was much more likely for transactions to fail with a penalty; the clients completed the other parts of their transactions, but their tentative hold expired before they could finalise their booking of the resources. When semantic atomicity was used, as stated in the previous paragraph, many transactions failed at an earlier time when they could have succeeded later, which reduced the number that completed successfully. However, as when a traditional scheme with no time out was used, transactions only ever failed without penalty.

Thus, traditional transaction techniques were shown to be much better at ensuring successful transaction completion utilising as many resources as possible. However, by considering the average duration of a transaction, it can be seen why they are not preferred for transactions in a service-oriented environment. Forcing clients to wait for a guarantee
that the requested resources will be available means that clients spend a lot of their transaction time simply waiting. When a traditional scheme is used, the average duration of a transaction is over 1200 time units, compared to only 228 when tentative hold is used, or 134 when semantic atomicity is offered. In many cases, clients would not be willing to wait for such a long time, and thus would not use the provider offering the traditional scheme.

### 7.2 Supporting Dynamic Transaction Levels

The previous scenario showed that using non-traditional transactional concepts such as tentative holds and semantic atomicity greatly reduces the average amount of time it takes for a client to complete a transaction. However, when tentative holds are used, clients are much more likely to experience transactions that end in a partially completed state, meaning that the risk for the client is increased. Alternately, when semantic atomicity is used, the risk to the client is again reduced, but the provider may turn away a client it could have supported if another client has a hold on the required resources which it later cancels. A potential way to reduce these risks is to allow providers to offer a dynamic level of transaction support; rather than the offered level of transaction support being an integrated part of the service, it becomes a part of a call to the service, potentially changing for each client request. In this way, a provider can choose the level of transaction support to offer at run-time, determining what level of risk (if any) it is willing to take in order to entice the client to use the service being offered. Clients can then determine whether the offered level of support is acceptable before deciding whether they should use that service or look for an acceptable alternative.

The scenario presented in this section examines how allowing dynamic levels of transaction support affects both clients and service providers. Since offering only a fixed level results in the level of risk experienced by the providers and clients to remain fairly constant for all transactions, any imbalance will be compounded, so that either the provider will be significantly worse off, or many clients disadvantaged. Dynamic levels allow any imbalance to be managed, which is shown to give fairer results for all involved parties.
7.2.1 Scenario Description

This scenario is based on the experiment described in [99]. There are three competing providers offering a particular resource, with each competing service being identical except for the offered level of transaction support. Each provider offers one of the following three levels of transaction support:

- **Semantic atomicity.** A client sends a request to the provider for a certain number of resources. If the provider offers these resources to the client then those resources cannot be offered to any other client. The client can, however, later decide to cancel the order without penalty, in which case the provider is again able to offer those resources to other clients.

- **Tentative hold.** A client sends a request to the provider asking if it could fulfil a particular order. The provider sends back a message indicating whether the requested number of resources are currently available. As long as the client does not finalise the booking, the provider is able to use the requested resources for other clients’ requests. If the resources become unavailable before the client finalises its booking, the provider lets the client know that the hold is no longer valid. Once a client has finalised an order, it cannot be cancelled without penalty.

- **Variable.** The provider offers semantic atomicity while its available resources are plentiful, but switches to only supporting tentative holds when the number of resources it can provide are reduced to a pre-set level.

The clients in this scenario have a set of actions to perform, including booking of some of the resources offered by the three competing providers. The booking of those resources is modelled using three separate providers each with their chosen level of transaction support. The rest of the actions in the transaction are modelled as a single activity that takes between 20 and 500 time units to complete. This action is set to fail 20% of the time. As in the previous scenario, most costs are set to 0, except for the prepare and commit costs (set to 1), and success budgets are set to 2. Further, for this scenario, the cost to compensate is set to −1 and client failure budgets are 1, except for some clients which are set to require semantic atomicity (and thus have a failure budget of 0). To allow support for tentative holds, clients are set to prefer enquire operations if possible in a client interaction, and only commit when an enquiry is not possible.
The behaviour of a client depends on the risks that client is willing to take, and the level of transaction support offered by the service providers. Some clients will be extremely risk adverse, not willing to use any provider that will not offer them semantic atomicity. Others will not care about the level of transaction support offered, and will be willing to use whatever a provider offers to them. However, given that the only difference between the various service providers is the level of transaction support they are offering, it is fair to assume that most clients will prefer to use a service offering a higher level of transaction support over one that offers a lower level. In the scenarios in this section, it is assumed that 80% of clients prefer a better level of transaction support (and contact each provider to determine the level of support that each will provide), 10% do not care what level of transaction support they are offered (and thus choose a single provider to contact initially, only contacting one of the others if the first cannot support their request), and 10% require semantic atomicity. These numbers are adequate for testing the viability of the variable transaction scheme, though are not necessarily realistic.

If a client cannot find a provider offering the service with an acceptable level of transaction support, then the client’s transaction must fail without penalty. Otherwise, the client can continue the transaction utilising the level of support offered by the provider. If that level is semantic atomicity, the client can book the resources immediately and then attempt to complete the rest of their transaction. If the rest of the transaction fails then the client can compensate the booking to finish without penalty. If, on the other hand, the rest of the transaction succeeds, then the client’s transaction ends successfully once the client finalises its booking.

A client’s actions when the chosen provider offers tentative hold are slightly different. Clients would, of course, prefer for their transaction to succeed, but, when tentative hold is offered, can choose whether they would prefer to fail by having the booking of resources succeed and the rest of the transaction fail, or having the rest of the transaction succeed but the booking of resources fail. In the scenarios in this section, half of all clients choose the former option, and half the latter. If the first option is chosen then the client books the resources immediately and then attempts the other actions in its transaction. If the other actions succeed then the overall outcome is success. However, if the other actions fail then the client has a partially completed transaction, with the resources booked and unable to be unbooked even though the rest of the transaction failed. On the other hand, if the
second option is chosen, the client requests a tentative hold on the items and then attempts to complete the rest of the transaction. If the rest of the transaction completes successfully then the client attempts to finalise their booking of the resources. If the resources can be booked successfully then the transaction completes successfully. Otherwise the tentative hold has been invalidated, making it impossible for the client to book those resources, between when the client attempted to complete the other actions and when the other actions completed successfully, resulting in a partially completed transaction.

This scenario was run three times with the number of resources being provided or requested changing for each. For the first two runs, each client requested to book between one and ten resources (chosen randomly). In the third run, half of the clients requested between one and ten resources, and the rest requested between 40 and 60 resources. For the first run, each provider offered only 1000 resources. This was far fewer than required by all clients, so resources were extremely limited. In the second run each provider offered 3500 resources, so there were enough resources for each client request. For the final run, in which clients request a larger number of resources, each provider had 7000 resources on offer. As in Section 7.1, the particular values chosen for this scenario’s parameters are reasonable assumptions that could represent a real-world scenario.

The parameters chosen for this scenario were selected to ensure adequate resource contention. The different levels of transaction support can only be properly tested when multiple clients attempt to access the same resources at the same time. If there are plentiful resources, the different client workflows can access the resources they respectively require without affecting any other concurrent workflow. Similarly, if there are limited resources but few client transactions run concurrently, the first workflows would successfully utilise the resources, leaving no resources for later transactions, forcing the later workflows to fail. Further, it is necessary to make the other action modelled in the scenario fail occasionally to ensure that a client workflow can fail even when the resources required from a particular service in the workflow are available.

7.2.2 Results

The results of each run of this scenario can be seen in Tables 7.2, 7.3, and 7.4, respectively. In each table, the first column indicates the level of transaction support offered by each of the three providers. An “S” indicates the provider supported semantic atomicity, a “T”
that it supported tentative hold, and a “V” that it supported the variable scheme. Thus “STV” indicates that the first provider (P1) offered semantic atomicity, the second provider (P2) offered tentative hold, and the third provider (P3) offered the variable scheme. The next columns show the utility of each provider, that is the percentage of those resources it initially had available that were booked by a client when all transactions had completed. The final columns show the outcomes of the clients in the scenario. The “Successful” column shows the percentage of clients that had their transaction completed successfully, the “Fail Without Penalty” column indicates how many clients ended in a Failed state with zero cost, “Fail With Resource Penalty” shows the percentage of clients who successfully booked the resources but had the rest of their transaction fail (and thus had to undo the booking), and “Fail With Other Penalty” shows the percentage of clients that were unable to book the resources but had the rest of the transaction complete successfully (which they then had to undo).

The results of the first run, in which providers had limited resources, are in Table 7.2.

As the number of resources was so limited, provider utility was always fairly high (above 90%), however it can again be seen that utilising semantic atomicity reduces a provider’s utility compared to when tentative hold is used (this is because the risk for the clients is reduced, so fewer client transactions end in a partially completed state). Of greater significance for this run are the differences in client outcomes. The highest percentage (60.9%) of clients succeeded when all three providers utilised semantic atomicity, and no clients failed with penalty. In contrast, when only tentative hold was used, only 48.0% of

<table>
<thead>
<tr>
<th>Level Of Transaction Support</th>
<th>Provider Utility (%)</th>
<th>Client Outcome (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>SSS</td>
<td>95.5</td>
<td>95.0</td>
</tr>
<tr>
<td>SST</td>
<td>89.1</td>
<td>96.0</td>
</tr>
<tr>
<td>SSV</td>
<td>94.7</td>
<td>96.6</td>
</tr>
<tr>
<td>STT</td>
<td>95.1</td>
<td>100.0</td>
</tr>
<tr>
<td>STV</td>
<td>92.7</td>
<td>100.0</td>
</tr>
<tr>
<td>SVV</td>
<td>98.5</td>
<td>99.6</td>
</tr>
<tr>
<td>TTT</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>TTV</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>TVV</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>VVV</td>
<td>100.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Table 7.2: Scenario 2 with limited resources
client transactions completed successfully, and 33.5% failed with some sort of penalty. This further shows that semantic atomicity offers better support for clients than tentative hold. All other results lay between these two extremes; more providers offering tentative holds led to a lower client success rate. When only the variable scheme was used (in which providers initially offer semantic atomicity but switch to offering tentative holds when the number of available resources drops), for example, 57.9% of transactions completed successfully and only 7.0% failed with penalty. Thus, offering the variable scheme significantly reduced the number of client transactions that failed with penalty, while reasonably improving client success rate.

The results of the second run, in which providers offered sufficient resources for clients, are shown in Table 7.3. Firstly, there were sufficient resources for every client, so transactions could only fail with penalty if the resources were booked and the rest of the transaction failed. This can be seen by the fact that the “Fail With Other Penalty” column is always 0% for this run. Otherwise, the client outcomes agree with results from the previous run; more transactions finished successfully when better transactional support was offered. In fact, when only semantic atomicity and the variable schemes were offered, the providers offering the variable level never reached the change-over threshold, so the results are identical to when only semantic atomicity was offered. Again, the more providers that offered tentative holds, the higher both the overall failure rate and the failure with penalty rate.

Provider utility is much more interesting in this run. When all providers offered the

<table>
<thead>
<tr>
<th>Level Of Transaction Support</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>Successful</th>
<th>Fail Without Penalty</th>
<th>Fail With Resource Penalty</th>
<th>Fail With Other Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSS</td>
<td>40.5</td>
<td>37.8</td>
<td>37.6</td>
<td>80.8</td>
<td>19.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>SST</td>
<td>55.5</td>
<td>56.0</td>
<td>4.3</td>
<td>79.5</td>
<td>20.1</td>
<td>0.4</td>
<td>0.0</td>
</tr>
<tr>
<td>SSV</td>
<td>40.5</td>
<td>37.8</td>
<td>37.6</td>
<td>80.8</td>
<td>19.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>STT</td>
<td>85.8</td>
<td>15.9</td>
<td>13.9</td>
<td>76.6</td>
<td>19.9</td>
<td>3.5</td>
<td>0.0</td>
</tr>
<tr>
<td>STV</td>
<td>57.1</td>
<td>2.7</td>
<td>52.3</td>
<td>77.9</td>
<td>21.7</td>
<td>0.4</td>
<td>0.0</td>
</tr>
<tr>
<td>SVV</td>
<td>40.5</td>
<td>37.8</td>
<td>37.6</td>
<td>80.8</td>
<td>19.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>TTT</td>
<td>41.3</td>
<td>38.5</td>
<td>36.2</td>
<td>70.2</td>
<td>17.5</td>
<td>12.3</td>
<td>0.0</td>
</tr>
<tr>
<td>TTV</td>
<td>17.5</td>
<td>15.3</td>
<td>85.9</td>
<td>78.0</td>
<td>17.6</td>
<td>4.4</td>
<td>0.0</td>
</tr>
<tr>
<td>TVV</td>
<td>4.8</td>
<td>57.2</td>
<td>52.4</td>
<td>78.3</td>
<td>21.1</td>
<td>0.6</td>
<td>0.0</td>
</tr>
<tr>
<td>VVV</td>
<td>40.5</td>
<td>37.8</td>
<td>37.6</td>
<td>80.8</td>
<td>19.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 7.3: Scenario 2 with sufficient resources
CHAPTER 7. VALIDATION

![Table 7.4: Scenario 2 with half of the clients requesting a large number of resources](image)

<table>
<thead>
<tr>
<th>Level Of Transaction Support</th>
<th>Provider Utility (%)</th>
<th>Client Outcome (%)</th>
<th>Fail Without Penalty</th>
<th>Fail With Resource Penalty</th>
<th>Fail With Other Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>Successful</td>
<td></td>
</tr>
<tr>
<td>SSS</td>
<td>86.4</td>
<td>88.2</td>
<td>89.3</td>
<td>73.5</td>
<td>26.5</td>
</tr>
<tr>
<td>SST</td>
<td>92.8</td>
<td>91.2</td>
<td>100.0</td>
<td>70.9</td>
<td>17.6</td>
</tr>
<tr>
<td>SSV</td>
<td>92.8</td>
<td>91.6</td>
<td>94.6</td>
<td>74.4</td>
<td>22.2</td>
</tr>
<tr>
<td>STT</td>
<td>96.8</td>
<td>100.0</td>
<td>100.0</td>
<td>69.6</td>
<td>17.3</td>
</tr>
<tr>
<td>STV</td>
<td>96.3</td>
<td>100.0</td>
<td>99.4</td>
<td>72.1</td>
<td>19.1</td>
</tr>
<tr>
<td>SVV</td>
<td>97.2</td>
<td>99.2</td>
<td>93.9</td>
<td>75.5</td>
<td>19.6</td>
</tr>
<tr>
<td>TTT</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>64.0</td>
<td>17.5</td>
</tr>
<tr>
<td>TTV</td>
<td>100.0</td>
<td>100.0</td>
<td>99.5</td>
<td>72.3</td>
<td>15.5</td>
</tr>
<tr>
<td>TVV</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>71.9</td>
<td>18.3</td>
</tr>
<tr>
<td>VVV</td>
<td>100.0</td>
<td>99.5</td>
<td>99.6</td>
<td>74.4</td>
<td>20.1</td>
</tr>
</tbody>
</table>

Table 7.4: Scenario 2 with half of the clients requesting a large number of resources

same level of transaction support, each provider was used fairly equally. However, since the majority of clients preferred a higher level of transaction support, it can be seen that when some providers offered only tentative holds and others some form of semantic atomicity, then the ones offering semantic atomicity had a much higher utility. For example, when all three providers offered semantic atomicity, the utility of the providers was 40.5%, 37.8%, and 37.6%. If, however, one of the providers offered tentative hold, the utility for the two offering semantic atomicity was 55.5% and 56.0%, and the utility for the one offering tentative holds was only 4.3%. Similarly, when all providers offered tentative holds, the respective provider utilities were 41.3%, 38.5%, and 36.2%. When just one of these providers offered the variable scheme, its utility jumped to 85.9% while the others dropped to 17.5% and 15.3%. This is because the provider offering the variable scheme was used almost exclusively until it reached its change-over threshold, after which all three offered the same level of transaction support and thus were each equally likely to be chosen by a particular client. This shows that providing a higher level of transaction support than competitors can be extremely worthwhile for providers or, conversely, that offering a lower level than just one competitor can be quite costly.

Finally, Table 7.4 presents the results for when the providers offer 7000 resources and half of the clients request a large number of these. Once again, fewer transactions failed with penalty as more providers offered higher levels of transaction support. However, as many of the clients booked a large number of resources when semantic atomicity was offered (thus making those resources unavailable for other clients), the utility of providers
offering support for semantic atomicity was also reduced. For example, when all providers offered semantic atomicity, the average utility of a provider was 88.0%. Compare this to when all providers offered tentative hold, where the utility of each provider was 100.0%. Of course, when only tentative hold was offered, 18.5% of the transactions failed with a penalty. However, when each provider offered the variable scheme, average provider utility was significantly higher than when only semantic atomicity was used (99.7%), and failures with penalty were still fairly low (5.5%). Thus, the variable scheme offers a good compromise to help keep both providers and clients happy.

7.3 Summary

This chapter presented two scenarios to validate the dynamic transaction system introduced in Chapter 4 and the simulator described in Chapter 6. The first scenario confirmed results previously published in the literature, showing that use of traditional transaction schemes in service-oriented environments can greatly increase the duration of transactions being processed by the system, to the extent that the use of traditional schemes is often not viable. The use of semantic atomicity or tentative holds greatly reduces the average length of transactions, but also reduces the number of transactions that complete successfully. In the case of tentative holds, an extra risk is associated with the clients, because a tentative hold may become invalid at any time. Conversely, when semantic atomicity is used, clients have no increased risk, but providers may be forced to deny a service to one client and then have another client compensate an action which could have been used to process the other client’s request. These results verify that the simulator matches expected results, justifying the simulator’s use for comparing transaction schemes in service-oriented environments.

The second presented scenario demonstrated that allowing service providers to offer varying levels of transaction support for a single service can result in benefits for both service providers and clients. This scenario compared providers’ use of tentative holds or semantic atomicity to a varying scheme where the provider alternated between the two. The use of tentative holds led to more clients failing with penalty, and the use of semantic atomicity decreased provider utility whenever resources were highly contended. The varying scheme, therefore, allowed the providers to balance these two extremes and
provide increased support for clients without placing too large a burden on the providers. These results demonstrate that the dynamic transaction system presented in this thesis offers an improvement to the static levels of support currently available.
Chapter 8

Conclusion

If necessity is the mother of invention, it’s the father of cooperation. And we’re cooperating like never before.

John Ashcroft

Cooperation between specialised parties is often necessary to allow large tasks to be completed. Clients commonly need to combine services offered by multiple providers into a logically single action. In service-oriented architectures, where clients are able to individually call services from multiple providers, accidental cooperation can be achieved by placing the service calls required by a client into a workflow. However, if this implicit support for cooperation is the only support offered by providers in the workflow, then any error leaves the client with only part of an activity completed, despite the client viewing the entire activity as a single unit. Introducing support for transactions into a service-oriented environment gives clients access to the atomicity they require for their actions by allowing the providers to offer better support for cooperation.

Using existing transaction technology gives providers in the Web Services environment one of two choices: always offer a particular level of transactional support, or never offer transactional support. However, the environment in which software services live is constantly evolving [8]. Thus, the level of transactional support that a provider wishes to offer may vary over time. This thesis presents a method that allows providers to dynamically change the level of transaction support they are offering.

By deciding the given level of transactional support each time a service is called, providers are able to examine their current situation and adapt to offering a level of support with which they are comfortable. Clients can then combine the various guarantees offered
by the providers in the client’s workflow in such a way that the level of risk to the client is acceptable. Such a technique was discussed in a theoretical sense, and a possible mapping to real-world technologies was given.

To test the viability of such a system, a prototype model was developed. The results of experiments using this model revealed that allowing such dynamic transactional support can result in benefits for both providers and clients. The utility of providers can be increased by varying the level of transaction support offered for a particular service, and the ability of a client to calculate the risk of attempting a workflow means that the client can be assured the outcome of any attempt will always be, at least, acceptable.

The remainder of this chapter concludes the work by first revisiting the research questions asked in Chapter 1. This is followed by a discussion of possible future research directions in Section 8.2.

8.1 Research Questions

This section returns to the research questions presented in Chapter 1, examining how each has been handled in the work presented in this thesis.

8.1.1 Research Question 1

- *How has cooperation been achieved in the past and in current systems?* What are the current techniques and practices, and what can we learn from them?

Service-oriented architectures were designed to overcome some of the issues with existing distributed computing systems. Chapter 2 examined these issues and showed how Web Services can be used to avoid problems such as compatibility in a heterogeneous environment, by utilising the almost-ubiquitous Web infrastructure to ensure support, and retaining autonomy in a multidatabase system. The loose coupling of service-oriented systems also makes it much easier for disparate participants to offer connectivity without requiring the detailed knowledge of each others’ systems that technologies such as CORBA implicitly need.

From here, the concept of transactions was examined. Transactions combine multiple actions into a logically single action with well-defined behaviour, even in the case of errors. Chapter 3 examined how transactions have been brought into the Web Services
environment, specifying the various standards and extensions that have been defined. The traditionally-required ACID properties were shown to not always be appropriate in the Web Services environment, with various reductions in the strength of these properties commonly being used.

The main problem identified with the existing standards was that they have limited support for deliberate cooperation between unconnected service providers in the quest for achievement of a client’s aims. The very nature of service-oriented systems implies accidental cooperation, and the current Web Services transactions standards extend this to also allow forced cooperation. Neither of these are ideal. Accidental cooperation leaves the client vulnerable in the case of failures, and the forced cooperation required in existing systems may push providers to offer a different level of cooperation from the level they would prefer to offer.

8.1.2 Research Question 2

- What levels of support can be offered to facilitate cooperation between multiple services? What patterns can a service provider support, in addition to performing their actual service, to help clients achieve their goals?

The various possible levels of transaction support were examined in Chapter 4 and Appendix B. These identify five basic operations: enquire, prepare, commit, compensate, and callback, and the concept of resilience. These basic operations can be combined in various ways to allow different levels of support for cooperation, and resilience can decrease the number of errors that occur by retrying failed actions whenever possible. By combining the five basic operations, providers have a much greater choice of the level of cooperation they wish to support than is typically available in current systems.

The enquire operation allows a client to determine whether a call would currently succeed without receiving any guarantee that any future request will achieve the same result. This is useful to help a client determine whether a particular service call is a viable part of a workflow. The prepare operation provides a guarantee, potentially with a time limit, that the client’s request will succeed. This reduces the client’s risk by providing the client the ability to ensure the rest of its workflow will complete successfully before actually requesting that the service be completed. The commit operation actually performs the client’s action. If a compensate operation is supported, the client is able to later undo a
previously completed service call, possibly at a price.

When a provider supports the callback operation, it removes the need for a client to poll the service provider to see if the provider’s situation has changed. A callback after an enquire operation can be used to support tentative holds if the callback is sent after a successful response to the initial enquiry. Otherwise, after a failed enquire, prepare, or commit operation, a callback indicates that the call may now succeed.

Resilience can be supported through the use of abstract services which contain a list of other providers that can be used to supply the required service. The abstract service acts as a proxy for the client, contacting one of the listed providers and requesting that it perform the action. If the contacted provider fails to perform the service, the abstract service can attempt to contact one of the alternate providers. In this way, the abstract service only notifies the client of any errors if none of the alternatives can perform the required service. This resilience is thus transparent to both the clients utilising the service and the providers listed in the abstract service.

8.1.3 Research Question 3

- How can providers choose the level of support for cooperation they wish to offer?

Given the possible patterns, how can a provider choose and advertise the subset of patterns it wishes to support?

While largely an individual decision by providers, Chapter 4 examined some criteria that providers may use to determine the level of transaction support they would be willing to offer for a particular service call. For example, information such as the current resources that the provider has available, historical information of how the service has been used, or the typical interaction of specific clients can all be used to help a provider decide on a level to support. A very important consideration, however, is any existing transactional guarantees that the provider has with other clients. This issue was examined in Chapter 5, which presented a method that providers can use to ensure any offered transactional guarantees are compatible with the provider’s current state.

Once the provider has decided on the level of transaction support it wishes to offer a client, it is necessary to reach an agreement with the client on that level of support. Chapter 4 and Appendix A examined and defined some Web Services standards that allow the exchange and agreement of such information. Further, the suggested methods
allow negotiation between service providers and clients to help ensure that an acceptable level for both parties can be found, if possible.

8.1.4 Research Question 4

- *How can the different levels of support offered by service providers be combined?*

If the different providers that a client wishes to use are offering different levels of support for cooperation, are they compatible, and what guarantees do they give? In particular, can a client make risk-based judgements as to whether or not it should proceed?

Chapter 5 presented a formal model, based on $\pi$-calculus, that allows a client to reason about its workflow. Given a success and a failure budget, the client can combine the different levels of transaction support offered by the providers in the workflow to choose a set of actions that guarantees, in the case of successful completion of the workflow, that the cost will not be greater than the success budget, or, in the case the workflow completes unsuccessfully, that the cost will not exceed the failure budget. The model allows the client significant freedom to prioritise the actions required to complete the workflow, but still attempts to determine, as quickly as possible, when a workflow cannot possibly complete successfully. This allows the client to specify the level of risk it is willing to accept, and then calculate possible actions that ensure an acceptable outcome for the client’s workflow.

8.1.5 Research Question 5

- *Does a system that allows providers to dynamically decide on the level of cooperation offered for a particular service, and clients to combine services with different guarantees, give better results than are currently achievable? How can the benefits of such a system be measured?* A prototype implementation is required to show that the theoretical results lead to a practical solution. Furthermore, how would such a solution be implemented using current standards and technologies?

Chapter 6 described a simulator to allow the effective comparison of the dynamic transaction system to the currently available systems. The simulator models transaction flow rather than service flow, ensuring the important aspects of a Service-Oriented environment are modelled while allowing complete control over all other aspects.
CHAPTER 8. CONCLUSION

The first experiment described in Chapter 7 shows that traditional transaction techniques are not always appropriate in a service-oriented environment. While traditional ACID schemes do guarantee results no worse than schemes such as tentative hold or semantic atomicity, they do so by greatly increasing the average running time of a transaction. This is typically not desirable in service-oriented environments, where transactional workflows can run for days or weeks; one such long-running transaction could force all other transactions, no matter how short, to run for that same long period of time.

The second experiment showed that offering a variable level of transactional support can be beneficial for both providers and clients. Having providers use a simple strategy to change between offering semantic atomicity and tentative holds significantly reduced the number of transactions that failed with penalty, compared to providers that offered only tentative holds. Further, offering the variable scheme increased provider utility because clients preferred to use a provider with better transaction support. While these benefits also held for providers which offered only semantic atomicity rather than the variable or tentative hold schemes, when more clients requested a larger number of resources, providers offering only semantic atomicity had a lower utility than the providers offering the variable scheme. Thus, the variable scheme works well for both providers and clients in all scenarios tested in this experiment. In particular, this verifies that there are situations in which offering a dynamic level of transaction support is beneficial for both clients and providers.

A real-world implementation of the dynamic transaction system has not been completed for this thesis. However, the negotiation and interaction phases could be implemented by adding support for the proposed enhancements to current standards. The negotiation phase could be implemented using the WS-Agreement-Negotiation [129] standard with the Transaction Description Language defined in Appendix A. Once the level of transaction support was agreed, the interaction could continue using the WS-Transaction standards [93, 91, 92] together with the new agreement coordination protocols introduced in Appendix B. A provider offering such a service could then potentially charge a fee to clients beyond that required by the service providers required to complete the client’s workflow.
8.2 Future Directions

This work presented a scheme that allows providers to dynamically choose the level of transaction support to offer for a particular service call. While factors that influence the choices made by a provider were presented, further investigation of the strategies that could be utilised to make the final decision is required. Similarly, the consideration of negotiation with clients, and altering of acceptable levels of transactional support to offer based on client responses, is possible but has not yet been fully investigated.

Coinciding with provider negotiation of transaction levels, is negotiation from the client side. Each client is different, and will thus have unique requirements when contacting any particular provider. Communicating these requirements in a way that maximises the likelihood of a provider agreeing to offer a matching service is a difficult problem. From a technical viewpoint, the techniques presented in this work allow this kind of negotiation, but effective use of these provisions requires the consideration of theories from areas such as economics, psychology, and game theory.

In addition to negotiation between a client and a single provider, clients must also consider negotiation between all parties involved in the client’s workflow. While the described system ensures the client is willing to accept the outcome of any workflow it begins, the client may wish to optimise that outcome. For example, when using services for which the client must pay an amount of money, the client may wish to minimise the amount required in either the best- or worst-case scenario. The prototype model described in Chapter 6 could be used to test such strategies.

The presented system assumed that each client included a transaction coordinator that was responsible for: handling negotiation of transaction levels; monitoring the client’s workflow; and interacting with specific providers. However, it would also be possible to define a coordinator service that could be utilised by multiple clients. Such a service could be implemented as an enhancement to the coordinator defined in the OASIS WS-TX standards [91, 92, 93] by adding support for the negotiation of transaction levels and using the protocols specified in Appendix B. The use of a coordinator service would reduce the processing requirements of clients and may allow optimisations based on the coordinator’s knowledge of multiple running transactions.

The simulator presented in this thesis was used to demonstrate that allowing providers
CHAPTER 8. CONCLUSION

to offer a dynamic level of transaction support can be beneficial to both providers and clients. The simulator could also be used to examine beneficial transaction strategies for a particular provider (or providers), based on the environment in which it is operating. Different transaction strategies could be tested by modelling available service providers and expected client strategies, and modifying only the particular provider’s strategy. Any alteration in results could then be directly attributable to the provider’s altered strategy, making it possible to determine a preferred strategy for that provider. Similarly, different client transaction strategies could be tested by altering how some of the clients determine which action they will next perform at any stage of their workflow. Results of these studies could then lead to implementation of particular strategies in the real-world environment to allow better use of the system by both clients and service providers. In particular, participants may be able to determine break-even points to ensure that they always choose the optimal strategy.
Appendix A

Specification Of Transaction Description Language

This document specifies the semantics and structure of the Transaction Description Language (TDL), Version 1.0. TDL is used to describe a workflow and the transactional properties required for that workflow. Below is a normative XML schema to allow such requirements to be expressed as a set of XML elements.

A.1 Notational Conventions And Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” are to be interpreted as described in RFC-2119 [19].

Pseudo schemas, similar to those in [9], are provided for each component. They use BNF-style conventions for XML entities: “?” denotes zero or one occurrences; “*” denotes zero or more occurrences; “+” denotes one or more occurrences. Attributes are assigned a value which corresponds to their type.

This specification uses namespace prefixes throughout; these are listed in Table A.1.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
</tr>
<tr>
<td>tdl</td>
<td><a href="http://davidjohnpaul.com/transaction/2010/tdl">http://davidjohnpaul.com/transaction/2010/tdl</a></td>
</tr>
</tbody>
</table>

Table A.1: Namespace prefixes used throughout this specification
A.2 Description

The TDL is used to describe a workflow and the transactional requirements for completing that workflow. While complete specification of complex workflows is beyond the scope of this specification, it does allow simple service calls to be described, and allows extensions to support more complex behaviour. More important for this specification is the details of the transactional requirements for the workflow.

The transactional requirements are specified using four base operations: *enquire*, *prepare*, *commit*, and *compensate*. *enquire* allows the requester to determine whether its request could currently succeed, without giving any guarantee that such success will be possible in the future. It is also possible to have the provider *callback* the requester if ever the response to an enquiry changes. This is a *callback* if ever an enquiry failed and the situation changes so that the request would now succeed, or a *tentative hold* if the enquiry initially succeeded but may not do so now. *prepare* allows traditional transaction-like semantics; the requester can prepare the workflow, at which point the provider guarantees that the workflow will complete successfully, but no other actions will be able to see any consequences of the actions until the requester asks to commit the workflow. *commit* attempts to perform the workflow, the level of transactional guarantee depending on what other actions are being supported. Finally *compensate* allows the requester to undo a previously completed request.

By default, all that is provided is the ability to *commit*; the other operations require explicit support to be mentioned in the TDL description. Some of the operations also have a duration associated with them; once the specified duration of time passes, any guarantees provided by that operation may be forgotten (for complete semantics see Section A.3).

A.3 Core Element Set

The TDL core element set contains the semantics for elements that are defined for TDL 1.0. All elements MUST be supported by TDL 1.0 compliant systems.

A.3.1 General Structure

This section describes the general elements that provide the overall structure for a TDL document.
A.3.1.1 TransactionDefinition

This element describes a transaction and its requirements. It contains a TransactionDescription element. It is the root element of TDL.

A.3.1.1.1 Multiplicity

The multiplicity of this element is one.

A.3.1.1.2 Type

This is a complex type. It MUST support the following elements:

- TransactionDescription

A.3.1.1.3 Attributes

The following attributes are defined:

id  The id of the transaction definition document. This is defined as an “xsd:id” and is in the default namespace of the document. The id MAY be omitted.

A.3.1.1.4 Pseudo Schema

```xml
<TransactionDefinition id="xsd:ID">
  <TransactionDescription ... />
  <xsd:any##other/>*
</TransactionDefinition>
```

A.3.1.2 TransactionDescription

This element describes a transaction and its requirements. It contains WorkflowIdentification and TransactionProperties elements.

A.3.1.2.1 Multiplicity

The multiplicity of this element is one.
A.3.1.2.2 Type

This is a complex type. It MUST support the following elements:

- WorkflowIdentification
- TransactionProperties

A.3.1.2.3 Attributes

No attributes are defined.

A.3.1.2.4 Pseudo Schema

```xml
<TransactionDescription>
  <WorkflowDefinition ... />
  <TransactionProperties ... />
  <xsd:any##other/>*
</TransactionDescription>
```

A.3.2 Workflows

The WorkflowDefinition defines the workflow to be performed by the provider. As this can be quite complex, full support is beyond the scope of these specifications. A simple specification that allows multiple parallel service calls is provided. A language such as BPEL [88] could be used to extend this support.

A.3.2.1 WorkflowDefinition

This element describes the workflow to be performed for this TDL document.

A.3.2.1.1 Multiplicity

The multiplicity of this element is one.

A.3.2.1.2 Type

This is a complex type. It MUST support the following elements:

- ServiceCall
A.3.2.1.3 Attributes

No attributes are defined.

A.3.2.1.4 Pseudo Schema

```xml
<WorkflowDefinition>
  <ServiceCall ... >/*</ServiceCall>
  <xsd:any##other/>*</xsd:any##other>
</WorkflowDefinition>
```

A.3.2.2 ServiceCall

This element describes a single service call to the provider offering the transactional guarantee.

A.3.2.2.1 Multiplicity

The multiplicity of this element is zero or more.

A.3.2.2.2 Type

This is a complex type. It MUST support the following elements:

- ServiceParameter

A.3.2.2.3 Attributes

The following attributes are defined:

serviceName The URI of the service to be called.

A.3.2.2.4 Pseudo Schema

```xml
<ServiceCall serviceName="xsd:NCName">
  <ServiceParameter ... />*</ServiceParameter>
</ServiceCall>
```
A.3.2.3 ServiceParameter

This element describes a parameter to be passed to a service call.

A.3.2.3.1 Multiplicity

The multiplicity of this element is zero or more.

A.3.2.3.2 Type

This is a complex type. The value of the parameter is included in the body of this element.

A.3.2.3.3 Attributes

The following attributes are defined:

name  The name of the parameter.

A.3.2.3.4 Pseudo Schema

<ServiceParameter name="xsd:NCName">xsd:string</ServiceParameter>

A.3.3 Transaction Details

The transactional properties required or provided for the workflow defined in the document are specified in this section.

A.3.3.1 TransactionProperties

This element describes the transactional properties to be ensured for the workflow.

A.3.3.1.1 Multiplicity

The multiplicity of this element is one.

A.3.3.1.2 Type

This is a complex type. It MUST support the following elements:

- Enquire
APPENDIX A. SPECIFICATION OF TRANSACTION DESCRIPTION LANGUAGE

- Prepare
- Commit
- Compensate

A.3.3.1.3 Attributes

No attributes are defined.

A.3.3.1.4 Pseudo Schema

```xml
<TransactionProperties>
  <Enquire ... />?
  <Prepare ... />?
  <Commit ... />?
  <Compensate ... />?
  <xsd:any##other/>*
</TransactionProperties>
```

A.3.3.2 Enquire

This element describes whether and how enquiries as to whether the required workflow could be completed successfully are handled. In particular, it specifies whether tentative holds and callbacks are applicable.

A.3.3.2.1 Multiplicity

The multiplicity of this element is zero or one.

A.3.3.2.2 Type

This is a complex type.

A.3.3.2.3 Attributes

The following attributes are defined:

`supported` Whether enquiry is supported at all. Defaults to “false”. If “true”, the provider MUST offer the ability to enquire whether the requested workflow could currently
be completed successfully, though there is no need to guarantee that any repeated request would succeed.

**callbackSupported** Whether the entity responsible for running the workflow will call back after a failed enquiry if ever it becomes possible to complete the workflow. Defaults to “false”. If “true”, the provider MAY call back the requester after a failed enquiry to let the requester know that the situation has changed and the enquiry would now be successful, though there is no need to guarantee that any future request would succeed.

**tentativeHoldSupported** Whether the entity responsible for running the workflow will call back after after a successful enquiry if ever the tentative hold is at risk of becoming invalid. Defaults to “false”. If “true”, the provider MUST call back the requester after a successful enquiry if ever the situation changes. The provider MAY also call back if ever it thinks the situation is likely to change.

### A.3.3.2.4 Pseudo Schema

```xml
<Enquire supported="xsd:boolean"? callbackSupported="xsd:boolean"
    tentativeHoldSupported="xsd:boolean"?/>
```

### A.3.3.3 Prepare

This element specifies whether the prepare stage for the workflow is supported, whether callbacks are offered, and the behaviour to be taken if a time out occurs.

#### A.3.3.3.1 Multiplicity

The multiplicity of this element is zero or one.

#### A.3.3.3.2 Type

This is a complex type.

#### A.3.3.3.3 Attributes

The following attributes are defined:
**supported** Whether the prepare stage is supported at all. Defaults to “false”. If “true”, the provider MUST offer the ability to prepare the workflow, which guarantees that a request to commit will be successful until a time out occurs (i.e. once the duration expires; see below).

**duration** The length of time after a prepare message that the hold will remain valid for. If unspecified, the duration is assumed to be forever. After this duration the provider MAY choose to release any locks held and perform the default action after the prepare stage.

**callbackSupported** Whether the provider will callback after a failed prepare if ever the situation changes. Defaults to “false”. If “true”, the provider MAY call back the requester to let them know that a failed attempt to prepare could now succeed. In this case, the provider MUST guarantee that the requestor’s call to prepare would be successful until a time out occurs (i.e. once the call back duration expires; see below), or the requester indicates that it no longer wishes to perform the workflow.

**callbackDuration** The length of time after the provider sends a call back message that the provider MUST guarantee the requester’s call to prepare will succeed. If unspecified, the duration is assumed to be forever. After this duration the provider MAY release any locks and continue as though it had never been called to prepare.

**commitOnTimeout** Whether to commit the workflow after a successful prepare step when a time out occurs (when the value is “true”), or to cancel (when the value is “false”). Defaults to “false”.

**A.3.3.3.4 Pseudo Schema**

```
<Prepare supported="xsd:boolean"? duration="xsd:duration"?
callbackSupported="xsd:boolean" callbackDuration="xsd:duration"?
cancelOnTimeout="xsd:boolean"?/>
```

**A.3.3.4 Commit**

This element specifies how commits are handled; specifically, if any callbacks are supported after a failed attempt to commit.
A.3.3.4.1 Multiplicity

The multiplicity of this element is zero or one.

A.3.3.4.2 Type

This is a complex type.

A.3.3.4.3 Attributes

The following attributes are supported:

- **callbackSupported** Whether the provider will call back after a failed attempt to commit if ever the situation changes so that the request would have succeeded. Defaults to “false”. If “true”, the provider MAY call back the requester if ever a failed attempt to commit would not succeed. The provider MUST guarantee that the requestor’s call to commit would be successful until a time out occurs (i.e. once the call back duration expires; see below), or the requester indicates that it no longer wishes to perform the workflow.

- **callbackDuration** The length of time after the provider sends a call back message that the provider MUST guarantee the requestor’s call to commit will succeed. If unspecified, the duration is assumed to be forever. After this duration the provider MAY release any locks and continue as though it had never been called to commit.

A.3.3.4.4 Pseudo Schema

```xml
<Commit callbackSupported="xsd:boolean"? callbackDuration="xsd:duration"/>
```

A.3.3.5 Compensate

Specifies if and how the workflow may be compensated after a successful commit.

A.3.3.5.1 Multiplicity

The multiplicity of this element is zero or one.
A.3.3.5.2 Type

This is a complex type.

A.3.3.5.3 Attributes

The following attributes are supported:

**supported** Whether compensation is supported at all. Defaults to “false”. If “true”, the provider MUST offer the ability to compensate the performed workflow until a time out occurs (i.e. once the duration expires; see below), or the requester indicates that it no longer wishes to compensate the workflow.

**commitDuration** The length of time after a successful commit that the provider MUST guarantee the requester’s call to compensate will succeed. If unspecified, the duration is assumed to be forever. After this duration the provider MAY ignore any calls to perform compensation.

**cancelOnTimeout** Whether the provider should automatically perform compensation after a time out occurs. Defaults to “false”.

A.3.3.5.4 Pseudo Schema

```xml
<Compensate supported="xsd:boolean"? commitDuration="xsd:duration"?
cancelOnTimeout="xsd:boolean"?/>
```

A.4 TDL Normative Schema

This section contains the full normative XML Schema definition for TDL 1.0.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tdl="http://davidjohnpaul.com/transaction/2010/tdl"
targetNamespace="http://davidjohnpaul.com/transaction/2010/tdl"
elementFormDefault="qualified">
  <xsd:complexType name="TransactionDefinitionType">
    <xsd:sequence>
      <xsd:element/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
APPENDIX A. SPECIFICATION OF TRANSACTION DESCRIPTION LANGUAGE

```xml
<xsd:element ref="tdl:TransactionDescription"/>
<xsd:element name="id" type="xsd:ID" use="optional"/>
<xsd:element ref="tdl:TransactionProperties"/>
<xsd:element ref="tdl:WorkflowDefinition" minOccurs="0"/>
<xsd:element ref="tdl:ServiceCall" minOccurs="0"/>
<xsd:element ref="tdl:ServiceParameter" minOccurs="0"/>
<xsd:element ref="tdl:Enquire" minOccurs="0"/>
<xsd:element ref="tdl:Prepare" minOccurs="0"/>
<xsd:element ref="tdl:Commit" minOccurs="0"/>
```

APPENDIX A. SPECIFICATION OF TRANSACTION DESCRIPTION LANGUAGE

```xml
<xsd:element ref="tdl:Compensate" minOccurs="0"/>
<xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="EnquireType">
  <xsd:attribute name="supported" type="xsd:boolean" default="false"/>
  <xsd:attribute name="callbackSupported" type="xsd:boolean" default="false"/>
  <xsd:attribute name="tentativeHoldSupported" type="xsd:boolean" default="false"/>
</xsd:complexType>

<xsd:complexType name="PrepareType">
  <xsd:attribute name="supported" type="xsd:boolean" default="false"/>
  <xsd:attribute name="duration" type="xsd:duration" default="P0Y"/>
  <xsd:attribute name="callbackSupported" type="xsd:boolean" default="false"/>
  <xsd:attribute name="callbackDuration" type="xsd:duration" default="P0Y"/>
  <xsd:attribute name="cancelOnTimeout" type="xsd:boolean" default="false"/>
</xsd:complexType>

<xsd:complexType name="CommitType">
  <xsd:attribute name="callbackSupported" type="xsd:boolean" default="false"/>
  <xsd:attribute name="callbackDuration" type="xsd:duration" default="P0Y"/>
</xsd:complexType>

<xsd:complexType name="CompensateType">
  <xsd:attribute name="supported" type="xsd:boolean" default="false"/>
  <xsd:attribute name="commitDuration" type="xsd:duration" default="P0Y"/>
  <xsd:attribute name="cancelOnTimeout" type="xsd:boolean" default="false"/>
</xsd:complexType>

<xsd:complexType name="TransactionDefinitionType">
  <xsd:complexType name="tdl:TransactionDefinitionType"/>
</xsd:complexType>

<xsd:complexType name="TransactionDescriptionType">
  <xsd:complexType name="tdl:TransactionDescriptionType"/>
</xsd:complexType>

<xsd:complexType name="WorkflowDefinitionType">
  <xsd:complexType name="tdl:WorkflowDefinitionType"/>
</xsd:complexType>

<xsd:complexType name="ServiceCallType">
  <xsd:complexType name="tdl:ServiceCallType"/>
</xsd:complexType>

<xsd:complexType name="ServiceParameterType">
  <xsd:complexType name="tdl:ServiceParameterType"/>
</xsd:complexType>

<xsd:complexType name="TransactionPropertiesType">
  <xsd:complexType name="tdl:TransactionPropertiesType"/>
</xsd:complexType>

<xsd:element name="Enquire" type="tdl:EnquireType"/>
<xsd:element name="Prepare" type="tdl:PrepareType"/>
<xsd:element name="Commit" type="tdl:CommitType"/>
<xsd:element name="Compensate" type="tdl:CompensateType"/>
</xsd:complexType>
</xsd:schema>
```
Appendix B

Specification Of Transaction Protocol

This appendix defines numerous Business Activity agreement coordination protocols for use with the MixedOutcome Business Activity coordination type defined in WS-BusinessActivity [92].

B.1 Protocols

This section defines the agreement coordination protocols defined for the dynamic transaction system. They are designed to replace or augment the BusinessAgreementWithCoordinatorCompletion protocol defined in WS-BusinessActivity [92]. This means that none of the actions occur until a “prepare” or “commit” message is sent by the coordinator. Similar protocols could be defined to replace the BusinessAgreementWithParticipantCompletion protocol for when the participant knows when to prepare or commit, but these are omitted for brevity.

B.1.1 Agreement Coordination Protocols

These protocols have the same preconditions as the BusinessAgreementWithCoordinatorCompletion protocol. Further, in all cases, both the coordinator and participant accept the “GetStatus” and “Status” messages defined in the OASIS standard. The returned status is the QName for the current state that the coordinator or participant is in, as described in the state diagrams in the below protocol.
specifications. Details such as the resending of messages or exchange of underlying protocol error messages are omitted. All parties must be prepared to receive duplicate notifications, which must be handled as specified below.

As with the WS-BusinessActivity standard, the coordinator may, after sending a message, receive a message that is consistent for the state before they sent the message. This should be handled by reverting to the old state and continuing from there. The one exception to this is on receipt of a “timeOut” message where the coordinator has proof that the message it sent was received before the time out occurred (which could be provided by using WS-Security timestamps [71] and WS-ReliableMessaging sequences [90]). In this case, the participant must move to the state the coordinator’s message would have taken it and continue as if the time out had never occurred.

B.1.1.1 Protocol Specifications

Below are the normative references for the protocols supported by this system.
B.1.1.1.1  Commit

Figure B.1 illustrates the abstract behaviour of this protocol using a state diagram\(^2\). Participants that register for this protocol must use the protocol identifier
\[\text{http://davidjohnpaul.com/transaction/Commit}\]

The messages accepted by the coordinator are:

- **committed**  This message indicates to the coordinator that the participant has completed all processing related to the protocol instance.

- **cannotComplete**  This message indicates to the coordinator that the participant has determined that it cannot successfully complete all processing related to the protocol instance. All completed work has been cancelled, and all pending work forgotten. The participant will not participate in any further work for the protocol.

- **exit**  This message indicates to the coordinator that the participant will no longer participate in the activity. All completed work has been cancelled, and all pending work forgotten.

- **cancelled**  This message indicates to the coordinator that the participant has successfully cancelled all processing related to the protocol instance.

- **fail**  This message indicates to the coordinator that the participant has failed and the state of work is unknown. This notification includes a QName defined in schema that indicates the cause of the failure.

The messages accepted by the participant are:

- **commit**  This message indicates to the participant that the coordinator would like to complete the activity. If the participant completes the activity successfully, it must reply with a “committed” message. Otherwise it must respond with a “cannotComplete” message.

- **notCompleted**  This message indicates to the participant that the coordinator is aware that the participant cannot successfully complete all processing related to the protocol instance.

---

1\(^1\)This would only occur if the participant maliciously chose to ignore the message from the coordinator.

2\(^2\)In all state diagrams, coordinator-generated messages are indicated with a solid arrow, and participant-generated messages with a dashed arrow.
exited  This message indicates to the participant that the coordinator is aware that the participant will no longer be participating in the activity.

cancel  This message indicates to the participant that the work being done must be cancelled, and all pending work forgotten.

failed  This message indicates to the participant that the coordinator is aware of a failure and no further actions are required of the participant.

B.1.1.1.2  CommitCompensate

This protocol is identical to that in Section B.1.1.1.1, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/CommitCompensate

- From the Committing state, the “committed” message goes to the Committed state in the state diagram in Figure B.2 rather than the Ended state.

- The coordinator accepts the additional messages:

closed  This message indicates to the coordinator that all processing related to the protocol instance has been successfully completed.

compensated  This message indicates to the coordinator that the participant has compensated all processing related to the protocol instance.
timeOut This message indicates to the coordinator that a time out has occurred, and the default action of the protocol will take place.

- The participant accepts the additional messages:

close This message indicates to the participant that the protocol instance is to be ended successfully.

compensate This message indicates to the participant that the work being done should be compensated.

timedOut This message indicates to the participant that the coordinator has acknowledged the time out and the default action of the protocol should occur.

B.1.1.1.3 CommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.2, except:

- Participants that register for this protocol must use the protocol identifier

http://davidjohnpaul.com/transaction/CommitPresumedCompensate

- From the CommittedTimeOut state, the “timedOut” message goes to the Compensating state rather than the Closing state.

B.1.1.1.4 CommitCallback

This protocol is identical to that in Section B.1.1.1.1, except:
• Participants that register for this protocol must use the protocol identifier

http://davidjohnpaul.com/transaction/CommitCallback

• From the Committing state, the “cannotComplete” message goes to the CommitFailed state in the state diagram in Figure B.3 rather than the NotCompleting state.

• The coordinator accepts the additional messages:

   callback This message indicates that the request made by the coordinator which previously could not be completed can now be completed.

   timeOut This message indicates to the coordinator that a time out has occurred, and the default action of the protocol will take place.

• The participant accepts the additional messages:

   awaitingCallback This message indicates to the participant that the coordinator has acknowledged the failed attempt and will not attempt the call again unless a “callback” is received.

   timedOut This message indicates to the participant that the coordinator has acknowledged the time out and the default action for the protocol should occur.

B.1.1.1.5 CommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.4, except:

• Participants that register for this protocol must use the protocol identifier

http://davidjohnpaul.com/transaction/CommitCallbackCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.2.

• From the Committing state, the “committed” message goes to the Committed state in Figure B.2 rather than the Ended state.
B.1.1.1.6 CommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.5, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/CommitCallbackPresumedCompensate

- From the CommittedTimeOut state, the “timedOut” message goes to the
  Compensating state rather than the Closing state.

B.1.1.1.7 PrepareCommit

This protocol is identical to that in Section B.1.1.1.1, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/PrepareCommit

- From the Active state, a “prepare” message from the coordinator moves to the
  Preparing state as defined in the state diagram in Figure B.4.

- The coordinator accepts the additional messages:
**Prepared** This message indicates to the coordinator that the request to “prepare” has succeeded and a future call to commit within the time out period is guaranteed to succeed.

**timeOut** This message indicates to the coordinator that a time out has occurred, and the default action of the protocol will take place.

- The participant accepts the additional messages:

**Prepare** This message indicates to the participant that the coordinator would like the participant to guarantee that a future call to commit will succeed. The participant MAY send a “prepared” message if it can guarantee that a call from the coordinator to commit will succeed until at least the specified time out period.

**timedOut** This message indicates to the participant that the coordinator has acknowledged the time out and the default action of the protocol should occur.

**B.1.1.1.8 PrepareCommitCompensate**

This protocol is identical to that in Section B.1.1.1.7, except:

- Participants that register for this protocol must use the protocol identifier

`http://davidjohnpaul.com/transaction/PrepareCommitCompensate`

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.2.

- From the Committing state, the “committed” message goes to the Committed state in Figure B.2 rather than the Ended state.

**B.1.1.1.9 PrepareCommitPresumedCompensate**

This protocol is identical to that in Section B.1.1.1.8, except:

- Participants that register for this protocol must use the protocol identifier

`http://davidjohnpaul.com/transaction/PrepareCommitPresumedCompensate`

- From the CommittedTimeOut state, the “timedOut” message goes to the Compensating state rather than the Closing state.
B.1.1.1.10 PrepareCommitCallback

This protocol is identical to that in Section B.1.1.1.7, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/PrepareCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.4.

- From the Committing state, the “cannotComplete” message goes to the CommitFailed state in the state diagram in Figure B.3 rather than the NotCompleting state.

B.1.1.1.11 PrepareCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.10, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/PrepareCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.2.

- From the Committing state, the “committed” message goes to the Committed state in Figure B.2 rather than the Ended state.

B.1.1.1.12 PrepareCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.11, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/PrepareCommitCallbackPresumedCompensate

- From the CommittedTimeOut state, the “timedOut” message goes to the Compensating state rather than the Closing state.
B.1.1.13 PreparePresumedCommit

This protocol is identical to that in Section B.1.1.7, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PreparePresumedCommit

- From the PreparedTimeOut state, the “timedOut” message goes to the Committing state rather than the Cancelling state.

B.1.1.14 PreparePresumedCommitCompensate

This protocol is identical to that in Section B.1.1.8, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PreparePresumedCommitCompensate

- From the PreparedTimeOut state, the “timedOut” message goes to the Committing state rather than the Cancelling state.

B.1.1.15 PreparePresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.14, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PreparePresumedCommitPresumedCompensate

- From the CommittedTimeOut state, the “timedOut” message goes to the Compensating state rather than the Closing state.

B.1.1.16 PreparePresumedCommitCallback

This protocol is identical to that in Section B.1.1.10, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PreparePresumedCommitCallback

- From the PreparedTimeOut state, the “timedOut” message goes to the Committing state rather than the Cancelling state.
B.1.1.1.17 PreparePresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.11, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PreparePresumedCommitCallbackCompensate

- From the PreparedTimeOut state, the “timedOut” message goes to the Committing state rather than the Cancelling state.

B.1.1.1.18 PreparePresumedCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.17, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PreparePresumedCommitCallbackPresumedCompensate

- From the CommittedTimeOut state, the “timedOut” message goes to the Compensating state rather than the Closing state.

B.1.1.1.19 PrepareCallbackCommit

This protocol is identical to that in Section B.1.1.1.7, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackCommit
• From the Preparing state, the “cannotComplete” message goes to the PrepareFailed state in Figure B.5 rather than the NotCompleting state.

• The coordinator accepts the additional messages:

   **callback** This message indicates that the request made by the coordinator which previously could not be completed can now be completed.

   **timeOut** This message indicates to the coordinator that a time out has occurred, and the default action of the protocol will take place.

• The participant accepts the additional messages:

   **awaitingCallback** This message indicates to the participant that the coordinator has acknowledged the failed attempt and will not attempt the call again unless a “callback” is received.

   **timedOut** This message indicates to the participant that the coordinator has acknowledged the time out and the default action for the protocol should occur.

**B.1.1.1.20 PrepareCallbackCommitCompensate**

This protocol is identical to that in Section B.1.1.1.19, except:

• Participants that register for this protocol must use the protocol identifier

  http://davidjohnpaul.com/transaction/PrepareCallbackCommitCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.2.

• From the Committing state, the “committed” message goes to the Committed state in Figure B.2 rather than the Ended state.

**B.1.1.1.21 PrepareCallbackCommitPresumedCompensate**

This protocol is identical to that in Section B.1.1.1.20, except:

• Participants that register for this protocol must use the protocol identifier

  http://davidjohnpaul.com/transaction/PrepareCallbackCommitPresumedCompensate

• From the CommittedTimeOut state, the “timedOut” message goes to the Compensating state rather than the Closing state.
B.1.1.1.22 PrepareCallbackCommitCallback

This protocol is identical to that in Section B.1.1.1.19, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.4.

- From the Committing state, the “cannotComplete” message goes to the CommitFailed state in the state diagram in Figure B.3 rather than the NotCompleting state.

B.1.1.1.23 PrepareCallbackCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.22, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.2.

- From the Committing state, the “committed” message goes to the Committed state in Figure B.2 rather than the Ended state.

B.1.1.1.24 PrepareCallbackCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.23, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackCommitCallbackPresumedCompensate

- From the CommittedTimeOut state, the “timedOut” message goes to the Compensating state rather than the Closing state.
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

B.1.1.1.25 PrepareCallbackPresumedCommit

This protocol is identical to that in Section B.1.1.1.19, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackPresumedCommit
- From the PreparedTimeOut state, the “timedOut” message goes to the Committing state rather than the Cancelling state.

B.1.1.1.26 PrepareCallbackPresumedCommitCompensate

This protocol is identical to that in Section B.1.1.1.20, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackPresumedCommitCompensate
- From the PreparedTimeOut state, the “timedOut” message goes to the Committing state rather than the Cancelling state.

B.1.1.1.27 PrepareCallbackPresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.26, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackPresumedCommitPresumedCompensate
- From the CommittedTimeOut state, the “timedOut” message goes to the Compensating state rather than the Closing state.

B.1.1.1.28 PrepareCallbackPresumedCommitCallback

This protocol is identical to that in Section B.1.1.1.22, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackPresumedCommitCallback
- From the PreparedTimeOut state, the “timedOut” message goes to the Committing state rather than the Cancelling state.
B.1.1.29 PrepareCallbackPresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.23, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackPresumedCommitCallbackCompensate

- From the PreparedTimeOut state, the “timedOut” message goes to the Committing state rather than the Cancelling state.

B.1.1.30 PrepareCallbackPresumedCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.29, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/PrepareCallbackPresumedCommitCallbackPresumedCompensate

- From the CommittedTimeOut state, the “timedOut” message goes to the Compensating state rather than the Closing state.

B.1.1.31 EnquireCommit

This protocol is identical to that in Section B.1.1.1, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCommit

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

- The coordinator accepts the additional messages:
enquirySuccessful This message indicates to the coordinator that the participant could currently complete the requested action if asked to do so, but gives no guarantee that any future request would similarly succeed.

enquiryFailed This message indicates to the coordinator that the participant could not currently complete the requested action if asked to do so, but gives no guarantee that any future request would similarly fail.

- The participant accepts the additional message:

enquire This message indicates that the coordinator would like to know if their request can currently succeed. If the participant could not successfully complete the requested action, the participant MUST respond with an “enquiryFailed” message. Otherwise, the participant MAY respond with an “enquirySuccessful” message.

B.1.1.1.32 EnquireCommitCompensate

This protocol is identical to that in Section B.1.1.1.2, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/EnquireCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.33 EnquireCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.3, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/EnquireCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

B.1.1.1.34 EnquireCommitCallback

This protocol is identical to that in Section B.1.1.1.4, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.35 EnquireCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.5, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.36 EnquireCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.6, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.
B.1.1.1.37 EnquirePrepareCommit

This protocol is identical to that in Section B.1.1.7, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.38 EnquirePrepareCommitCompensate

This protocol is identical to that in Section B.1.1.8, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.39 EnquirePrepareCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.9, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.
B.1.1.1.40  EnquirePrepareCommitCallback

This protocol is identical to that in Section B.1.1.1.10, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.41  EnquirePrepareCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.11, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.42  EnquirePrepareCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.12, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

B.1.1.1.43 EnquirePreparePresumedCommit

This protocol is identical to that in Section B.1.1.1.13, except:

- Participants that register for this protocol must use the protocol identifier

  http://davidjohnpaul.com/transaction/EnquirePreparePresumedCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.44 EnquirePreparePresumedCommitCompensate

This protocol is identical to that in Section B.1.1.1.14, except:

- Participants that register for this protocol must use the protocol identifier

  http://davidjohnpaul.com/transaction/EnquirePreparePresumedCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.45 EnquirePreparePresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.15, except:

- Participants that register for this protocol must use the protocol identifier

  http://davidjohnpaul.com/transaction/EnquirePreparePresumedCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.
B.1.1.1.46  EnquirePreparePresumedCommitCallback

This protocol is identical to that in Section B.1.1.1.16, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePreparePresumedCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.47  EnquirePreparePresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.17, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePreparePresumedCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.48  EnquirePreparePresumedCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.18, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePreparePresumedCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

B.1.1.1.49 EnquirePrepareCallbackCommit

This protocol is identical to that in Section B.1.1.1.19, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackCommit

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state
diagram in Figure B.6.

B.1.1.1.50 EnquirePrepareCallbackCommitCompensate

This protocol is identical to that in Section B.1.1.1.20, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackCommitCompensate

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state
diagram in Figure B.6.

B.1.1.1.51 EnquirePrepareCallbackCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.21, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state
diagram in Figure B.6.
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

B.1.1.1.52 EnquirePrepareCallbackCommitCallback

This protocol is identical to that in Section B.1.1.1.22, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.53 EnquirePrepareCallbackCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.23, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.54 EnquirePrepareCallbackCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.24, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.
B.1.1.1.55  EnquirePrepareCallbackPresumedCommit

This protocol is identical to that in Section B.1.1.1.25, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackPresumedCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.56  EnquirePrepareCallbackPresumedCommitCompensate

This protocol is identical to that in Section B.1.1.1.26, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackPresumedCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.57  EnquirePrepareCallbackPresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.27, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackPresumedCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.
APPENDIX B. SPECIFICATION OF TRANSACTION Protocol

B.1.1.1.58 EnquirePrepareCallbackPresumedCommitCallback

This protocol is identical to that in Section B.1.1.28, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackPresumedCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.59 EnquirePrepareCallbackPresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.29, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackPresumedCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.

B.1.1.1.60 EnquirePrepareCallbackPresumedCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.30, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquirePrepareCallbackPresumedCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.6.
B.1.1.61 EnquireCallbackCommit

This protocol is identical to that in Section B.1.1.1, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/EnquireCallbackCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

B.1.1.62 EnquireCallbackCommitCompensate

This protocol is identical to that in Section B.1.1.2, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/EnquireCallbackCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.
B.1.1.1.63 EnquireCallbackCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.3, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

B.1.1.1.64 EnquireCallbackCommitCallback

This protocol is identical to that in Section B.1.1.4, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

B.1.1.1.65 EnquireCallbackCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.5, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.
B.1.1.1.66 EnquireCallbackCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.6, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

B.1.1.1.67 EnquireCallbackPrepareCommit

This protocol is identical to that in Section B.1.1.7, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.68 EnquireCallbackPrepareCommitCompensate

This protocol is identical to that in Section B.1.1.8, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.
• From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.69 EnquireCallbackPrepareCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.9, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCommitPresumedCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

• From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.70 EnquireCallbackPrepareCommitCallback

This protocol is identical to that in Section B.1.1.10, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCommitCallback

• The coordinator and participant accept the additional messages defined in Section B.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

• From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.71 EnquireCallbackPrepareCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.11, except:

• Participants that register for this protocol must use the protocol identifier
The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.72 EnquireCallbackPrepareCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.12, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.73 EnquireCallbackPreparePresumedCommit

This protocol is identical to that in Section B.1.1.1.13, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/EnquireCallbackPreparePresumedCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

B.1.1.1.74 EnquireCallbackPreparePresumedCommitCompensate

This protocol is identical to that in Section B.1.1.1.14, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPreparePresumedCommitCompensate

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state
  diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which
  moves it into the Preparing state.

B.1.1.1.75 EnquireCallbackPreparePresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.15, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPreparePresumedCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state
  diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which
  moves it into the Preparing state.

B.1.1.1.76 EnquireCallbackPreparePresumedCommitCallback

This protocol is identical to that in Section B.1.1.1.16, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPreparePresumedCommitCallback

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.77  EnquireCallbackPreparePresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.17, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPreparePresumedCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.78  EnquireCallbackPreparePresumedCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.18, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPreparePresumedCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.79  EnquireCallbackPrepareCallbackCommit

This protocol is identical to that in Section B.1.1.1.19, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackCommit

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state
  diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which
  moves it into the Preparing state.

B.1.1.1.80  EnquireCallbackPrepareCallbackCommitCompensate

This protocol is identical to that in Section B.1.1.1.20, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackCommitCompensate

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state
  diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which
  moves it into the Preparing state.

B.1.1.1.81  EnquireCallbackPrepareCallbackCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.21, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.
• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

• From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.82 EnquireCallbackPrepareCallbackCommitCallback

This protocol is identical to that in Section B.1.1.1.22, except:

• Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackCommitCallback

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

• From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.83 EnquireCallbackPrepareCallbackCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.23, except:

• Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackCommitCallbackCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

• From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

B.1.1.1.84 EnquireCallbackPrepareCallbackCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.24, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.85 EnquireCallbackPrepareCallbackPresumedCommit

This protocol is identical to that in Section B.1.1.1.25, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackPresumedCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.86 EnquireCallbackPrepareCallbackPresumedCommitCompensate

This protocol is identical to that in Section B.1.1.1.26, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackPresumedCommitCompensate

200
• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

• From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.87 EnquireCallbackPrepareCallbackPresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.27, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackPresumedCommitPresumedCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

• From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.88 EnquireCallbackPrepareCallbackPresumedCommitCallback

This protocol is identical to that in Section B.1.1.28, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackPresumedCommitCallback

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.

• From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.89 EnquireCallbackPrepareCallbackPresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.29, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackPresumedCommitCallbackCompensate
- The coordinator and participant accept the additional messages defined in Section B.1.1.31.
- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.
- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.90 EnquireCallbackPrepareCallbackPresumedCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.30, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/EnquireCallbackPrepareCallbackPresumedCommitCallbackPresumedCompensate
- The coordinator and participant accept the additional messages defined in Section B.1.1.31.
- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.7.
- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.91 TentativeHoldCommit

This protocol is identical to that in Section B.1.1.1, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCommit
• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

B.1.1.1.92 TentativeHoldCommitCompensate

This protocol is identical to that in Section B.1.1.1.2, except:

• Participants that register for this protocol must use the protocol identifier

http://davidjohnpaul.com/transaction/TentativeHoldCommitCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

B.1.1.1.93 TentativeHoldCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.3, except:

• Participants that register for this protocol must use the protocol identifier

http://davidjohnpaul.com/transaction/TentativeHoldCommitPresumedCompensate
• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

B.1.1.1.94 TentativeHoldCommitCallback

This protocol is identical to that in Section B.1.1.1.4, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCommitCallback

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

B.1.1.1.95 TentativeHoldCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.5, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCommitCallbackCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

B.1.1.1.96 TentativeHoldCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.6, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCommitCallbackPresumedCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.
• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

B.1.1.1.97 TentativeHoldPrepareCommit

This protocol is identical to that in Section B.1.1.1.7, except:

• Participants that register for this protocol must use the protocol identifier

http://davidjohnpaul.com/transaction/TentativeHoldPrepareCommit

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

• From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.98 TentativeHoldPrepareCommitCompensate

This protocol is identical to that in Section B.1.1.1.8, except:

• Participants that register for this protocol must use the protocol identifier

http://davidjohnpaul.com/transaction/TentativeHoldPrepareCommitCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

• From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.99 TentativeHoldPrepareCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.9, except:

• Participants that register for this protocol must use the protocol identifier
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

http://davidjohnpaul.com/transaction/TentativeHoldPrepareCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.100 TentativeHoldPrepareCommitCallback

This protocol is identical to that in Section B.1.1.1.10, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPrepareCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.101 TentativeHoldPrepareCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.11, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPrepareCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.102 TentativeHoldPrepareCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.12, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/TentativeHoldPrepareCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.103 TentativeHoldPreparePresumedCommit

This protocol is identical to that in Section B.1.1.1.13, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/TentativeHoldPreparePresumedCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.104 TentativeHoldPreparePresumedCommitCompensate

This protocol is identical to that in Section B.1.1.1.14, except:

- Participants that register for this protocol must use the protocol identifier
  
  http://davidjohnpaul.com/transaction/TentativeHoldPreparePresumedCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.
• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

• From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.105 TentativeHoldPreparePresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.15, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPreparePresumedCommitPresumedCompensate

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

• From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.106 TentativeHoldPreparePresumedCommitCallback

This protocol is identical to that in Section B.1.1.1.16, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPreparePresumedCommitCallback

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

• From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
APPENDIX B. SPECIFICATION OF TRANSACTION PROTOCOL

B.1.1.1.107 TentativeHoldPreparePresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.17, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldPreparePresumedCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.108 TentativeHoldPreparePresumedCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.18, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldPreparePresumedCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.109 TentativeHoldPrepareCallbackCommit

This protocol is identical to that in Section B.1.1.1.19, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackCommit
• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

• From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

**B.1.1.1.110 TentativeHoldPrepareCallbackCommitCompensate**

This protocol is identical to that in Section B.1.1.1.20, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackCommitCompensate

  • The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

  • From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

  • From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

**B.1.1.1.111 TentativeHoldPrepareCallbackCommitPresumedCompensate**

This protocol is identical to that in Section B.1.1.1.21, except:

• Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackCommitPresumedCompensate

  • The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

  • From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

  • From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.112 TentativeHoldPrepareCallbackCommitCallback

This protocol is identical to that in Section B.1.1.1.22, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.113 TentativeHoldPrepareCallbackCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.23, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.114 TentativeHoldPrepareCallbackCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.24, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackCommitCallbackPresumedCompensate
• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

• From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

**B.1.1.1.115 TentativeHoldPrepareCallbackPresumedCommit**

This protocol is identical to that in Section B.1.1.1.25, except:

• Participants that register for this protocol must use the protocol identifier [http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackPresumedCommit](http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackPresumedCommit)

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

• From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

**B.1.1.1.116 TentativeHoldPrepareCallbackPresumedCommitCompensate**

This protocol is identical to that in Section B.1.1.1.26, except:

• Participants that register for this protocol must use the protocol identifier [http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackPresumedCommitCompensate](http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackPresumedCommitCompensate)

• The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

• From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.117 TentativeHoldPrepareCallbackPresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.27, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackPresumedCommitPresumedCompensate
- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.
- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.
- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.118 TentativeHoldPrepareCallbackPresumedCommitCallback

This protocol is identical to that in Section B.1.1.1.28, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackPresumedCommitCallback
- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.
- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.
- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.119 TentativeHoldPrepareCallbackPresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.29, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackPresumedCommitCallbackCompensate
The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

This protocol is identical to that in Section B.1.1.1.30, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldPrepareCallbackPresumedCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.8.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

This protocol is identical to that in Section B.1.1.1.1, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.
B.1.1.1.122 TentativeHoldCallbackCommitCompensate

This protocol is identical to that in Section B.1.1.1.2, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

B.1.1.1.123 TentativeHoldCallbackCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.3, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.
• From the Active state, the participant accepts the “enquire” message as in the state
diagram in Figure B.9.

B.1.1.1.124  TentativeHoldCallbackCommitCallback

This protocol is identical to that in Section B.1.1.1.4, except:

• Participants that register for this protocol must use the protocol identifier
http://davidjohnpaul.com/transaction/TentativeHoldCallbackCommitCallback

• The coordinator and participant accept the additional messages defined in Section
B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state
diagram in Figure B.9.

B.1.1.1.125  TentativeHoldCallbackCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.5, except:

• Participants that register for this protocol must use the protocol identifier
http://davidjohnpaul.com/transaction/TentativeHoldCallbackCommitCallbackCompensate

• The coordinator and participant accept the additional messages defined in Section
B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state
diagram in Figure B.9.

B.1.1.1.126  TentativeHoldCallbackCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.6, except:

• Participants that register for this protocol must use the protocol identifier
http://davidjohnpaul.com/transaction/TentativeHoldCallbackCommitCallbackPresumedCompensate

• The coordinator and participant accept the additional messages defined in Section
B.1.1.1.31.

• From the Active state, the participant accepts the “enquire” message as in the state
diagram in Figure B.9
B.1.1.1.127 TentativeHoldCallbackPrepareCommit

This protocol is identical to that in Section B.1.1.1.7, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.128 TentativeHoldCallbackPrepareCommitCompensate

This protocol is identical to that in Section B.1.1.1.8, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.129 TentativeHoldCallbackPrepareCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.9, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.130 TentativeHoldCallbackPrepareCommitCallback

This protocol is identical to that in Section B.1.1.1.10, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.131 TentativeHoldCallbackPrepareCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.11, except:

- Participants that register for this protocol must use the protocol identifier 
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.132 TentativeHoldCallbackPrepareCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.12, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.133  TentativeHoldCallbackPreparePresumedCommit

This protocol is identical to that in Section B.1.1.1.13, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPreparePresumedCommit

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state
  diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which
  moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which
  moves it into the Preparing state.

B.1.1.1.134  TentativeHoldCallbackPreparePresumedCommitCompensate

This protocol is identical to that in Section B.1.1.1.14, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPreparePresumedCommitCompensate

- The coordinator and participant accept the additional messages defined in Section
  B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state
  diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which
  moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which
  moves it into the Preparing state.
B.1.1.1.135 TentativeHoldCallbackPreparePresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.15, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPreparePresumedCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.136 TentativeHoldCallbackPreparePresumedCommitCallback

This protocol is identical to that in Section B.1.1.1.16, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPreparePresumedCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.137 TentativeHoldCallbackPreparePresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.17, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPreparePresumedCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.138 TentativeHoldCallbackPreparePresumedCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.18, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPreparePresumedCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.139 TentativeHoldCallbackPrepareCallbackCommit

This protocol is identical to that in Section B.1.1.1.19, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.140 TentativeHoldCallbackPrepareCallbackCommitCompensate

This protocol is identical to that in Section B.1.1.1.20, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.141 TentativeHoldCallbackPrepareCallbackCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.21, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.142 TentativeHoldCallbackPrepareCallbackCommitCallback

This protocol is identical to that in Section B.1.1.1.22, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.143 TentativeHoldCallbackPrepareCallbackCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.23, except:

- Participants that register for this protocol must use the protocol identifier 
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.144 TentativeHoldCallbackPrepareCallbackCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.24, except:

- Participants that register for this protocol must use the protocol identifier 
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.145 TentativeHoldCallbackPrepareCallbackPresumedCommit

This protocol is identical to that in Section B.1.1.25, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackPresumedCommit

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.146 TentativeHoldCallbackPrepareCallbackPresumedCommitCompensate

This protocol is identical to that in Section B.1.1.26, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackPresumedCommitCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.147  TentativeHoldCallbackPrepareCallbackPresumedCommitPresumedCompensate

This protocol is identical to that in Section B.1.1.1.27, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackPresumedCommitPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.148  TentativeHoldCallbackPrepareCallbackPresumedCommitCallback

This protocol is identical to that in Section B.1.1.1.28, except:

- Participants that register for this protocol must use the protocol identifier
  http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackPresumedCommitCallback

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
B.1.1.1.149 TentativeHoldCallbackPrepareCallbackPresumedCommitCallbackCompensate

This protocol is identical to that in Section B.1.1.1.29, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackPresumedCommitCallbackCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.

B.1.1.1.150 TentativeHoldCallbackPrepareCallbackPresumedCommitCallbackPresumedCompensate

This protocol is identical to that in Section B.1.1.1.30, except:

- Participants that register for this protocol must use the protocol identifier http://davidjohnpaul.com/transaction/TentativeHoldCallbackPrepareCallbackPresumedCommitCallbackPresumedCompensate

- The coordinator and participant accept the additional messages defined in Section B.1.1.1.31.

- From the Active state, the participant accepts the “enquire” message as in the state diagram in Figure B.9.

- From the EnquiryCalled state, the participant accepts a “prepare” message which moves it into the Preparing state.

- From the TentativelyHeld state, the participant accepts a “prepare” message which moves it into the Preparing state.
Bibliography

Cited on pages 27 and 28.

Cited on page 28.

Cited on page 51.

Cited on page 51.

Cited on pages 41, 45, and 49.


Cited on page 19.

Cited on page 41.

Cited on page 42.

Cited on pages 96, 104, and 106.

Cited on pages 104 and 106.

Cited on page 18.

Cited on page 153.


Cited on page 41.

Cited on page 120.

Cited on pages 37, 39, and 57.

Cited on pages 33 and 36.

Cited on pages 41, 45, 48, and 49.

Cited on page 49.

Cited on pages 16, 17, and 51.

Cited on pages 16, 17, 34, and 51.
Cited on page 16.

Cited on page 12.

Cited on pages 39 and 57.

Cited on page 16.

Cited on page 37.

Cited on page 19.

Cited on page 14.

Cited on page 25.
Cited on page 97.

Cited on page 120.

Cited on pages 38, 41, and 42.

Cited on page 129.

Cited on page 16.

Cited on pages 34 and 35.

Cited on pages 29 and 57.


Cited on pages 41, 45, 46, and 47.

Cited on pages 16 and 51.

Cited on page 2.

Cited on page 35.

Cited on page 15.

Cited on page 49.

Cited on page 24.

Cited on page 24.
Cited on page 121.

Cited on page 38.

Cited on page 49.

Cited on page 28.

Cited on page 97.

Cited on pages 16, 18, 51, 86, and 167.

Cited on page 30.

Cited on page 37.


Cited on page 15.

Cited on page 16.

Cited on page 14.

Cited on page 1.

Cited on page 1.

Cited on pages 19, 52, 68, and 156.

Cited on page 41.

Cited on pages 19, 51, 86, and 167.
Cited on pages 39, 57, 150, and 151.

Cited on pages 37, 39, 57, 80, 150, 151, and 166.

Cited on pages 39, 150, and 151.

Cited on page 14.

Cited on pages 28, 38, and 48.

Cited on pages 2, 11, and 12.

Cited on page 35.

Cited on pages 10, 41, and 42.


Cited on page 40.

Cited on page 129.

Cited on page 34.


Cited on pages 18 and 35.

Cited on page 15.


Cited on page 17.
Cited on page 19.

Cited on page 19.

Cited on pages 10 and 33.


Cited on page 85.

Cited on page 117.

Cited on page 15.

Cited on pages 20, 51, 52, 53, and 54.

[123] Yang, W., and Tang, S. A solution for web services transaction. In *The International Conference on Hybrid Information Technology (ICHIT’06)*
Cited on page 30.

Cited on pages 30 and 63.

Cited on page 30.

Cited on pages 41, 45, and 46.

Cited on page 29.

Cited on pages 74 and 150.