UNIVERSITY OF ZULULAND

Grid-based Utility Middleware Infrastructure for Distributed Services Provisioning

by

Johnson Seun Iyilade
BSc. Hons., MSc

A thesis submitted in fulfillment of the requirement
for the degree of Doctor of Philosophy

in the
Department of Computer Science,
Faculty of Science and Agriculture

April, 2010

Table of Contents

List of Figures

List of Tables

Declaration of Authorship

I, **Johnson Seun Iyilade**, declare that the thesis entitled: **Grid-based Utility Middleware Infrastructure for Distributed Services Provisioning** was written by me and the work contained here is my own, except where explicitly stated otherwise in the text with a list of references given in the Bibliography**.** The work has not been submitted for another degree or diploma at any university or other institute of tertiary education.

Signed: _____

Date: _____

Acknowledgements

First and foremost, I acknowledge the almighty God, the author and giver of life and wisdom, who provided me with grace and strength to complete this PhD thesis successfully.

Besides, a research work of this magnitude requires the support and contributions of a number of people, who I came across during my study and I deeply acknowledge their various roles:

My supervisor, Prof M.O. Adigun provided the right guidance and leadership that keeps me focused on the main goal of my PhD work. His comments and insightful contributions are invaluable in helping me stay on track.

I am grateful to Klaas Kabini, who worked under my mentorship for his MSc, for his invaluable contribution to the discussions on a number of ideas in this thesis and for offering implementation support. I also appreciate Yomi Otebolaku, Tarirai Chani, Yomi Ipadeola, Buthelezi Mcebo, Sicelo Sibiya, Ijeoma Mba, and other master students, who worked with me during the period of my PhD study at the University of Zululand. They have individually contributed to various research discussions and concepts that shaped this thesis.

My PhD colleagues, Edgar Jembere, Mudali Pregasen, Thulani Nyandeni, and Dr Reuben Aremu, Dr O.O Olugbara, Dr Obeten Ekabua, also provided the needed encouragement and atmosphere for intellectual discussions and exchange of ideas. I appreciate your contributions.

Many thanks to the staff of Computer Science Department for their friendship and encouragement in the course of my PhD research.

To my dear friend and brother, Adeniyi Oke, of SaskTel, Canada, I appreciate your useful suggestions and insights in my work.

I am indebted to my parents and family: Elder and Mrs Iyilade; Engr Segun Iyilade and family; Pastor Tunde Iyilade and family; and my darling sister Oluwaranti; thanks for all your moral support and care.

To my darling wife and daughters: Yemisi, Gloria and Sophia, who have to cope with my absence in the home to complete this thesis, I am forever grateful for your love, patience and encouragement at all times.

Finally, my PhD study was funded by the generous sponsorship that I received from the Center of Excellence for Mobile e-Services, through the financial contributions of Telkom, THRIP and NRF. I am grateful to you for this great privilege!

This thesis is dedicated to:

My parents, Elder & Mrs Emmanuel Iyilade,
whose vision for a good education inspires my soul;

My wife and daughters: Yemisi, Gloria, and Sophia –
your warmth embrace and unfailing love
motivates me to be all I could be.

Abstract

As Service Oriented Computing becomes the prevalent methodology for engineering distributed applications, service composition remains a key challenge to the reality of on-the-fly application composition using available services in an ecosystem. Current state-of-the-art service composition strategies are based on a predefined sequence of actions or workflow and are unsuitable when applied in dynamic and open settings due to their lack of scalability, fault-tolerance, and high bandwidth cost overhead. To address this problem, this thesis makes the case for a new flexible and adaptive strategy for service composition that is suitable for highly dynamic and open distributed services provisioning environments.

To support this thesis and demonstrate our solution approach, we propose **MINDS** – a **M**iddleware **I**nfrastructure for **D**istributed **S**ervice Provisioning as a platform to analyze and elaborate our service composition strategy. MINDS is based on model of business processes as a collaborative conversation among software agents; It employs late (runtime) binding to services and facilitates richer interactivity of services by enabling services to be active and aware of changes in the user and/or execution environment. We further formulate a new service provisioning life-cycle process for runtime, automated composition based on MINDS strategy. The new life-cycle process is not based on a predefined sequence of actions or workflow but rather on composition goal defined based on user request. To evaluate our strategy and demonstrate its utility and applicability, we implement an experimental prototype for e-Tourism Virtual Enterprise Collaboration as a case study. We also carry out empirical analysis and simulation experiments to compare the performance of MINDS with related strategies using *scalability*, *fault-tolerance* and *bandwidth cost optimization* as performance metrics.

The results of the simulation experiments shows that MINDS is scalable as it require lower service matchmaking and composition time when the number of services and tasks increase respectively. Moreover, in a sample of 80 tasks, 34 (representing 42.5%) met deadline when the centralized service agent used in existing strategy failed, while 55 (representing 68.75%) met deadline when decentralized service agents used in MINDS failed. Also, more tasks met deadline in MINDS when the service provider failed than the centralized service agent approach because the failure was discovered at a minimal time. Finally, the simulation results also indicated that MINDS optimize bandwidth by requiring less amount of data traffic on the network. In a sample of 41 participating services in composition, the overhead cost for data transmissions in both decentralized and centralized scheme were respectively, 136 Rands and 267 Rands.

We conclude that in view of the limitations of existing strategies for service composition in dynamic and open settings, a new flexible and adaptive middleware strategy as proposed in this thesis becomes imperative. Such a solution offers a more efficient composition strategy in terms of scalability, fault-tolerance and communication cost. It provides the crucial platform required in the actualization of the future "Internet of Services" towards on-the-fly composition and runtime binding of services.

Chapter 1

Introduction

1.1 Overview and Statement of the Problem

The ubiquity of the Internet and advances in communication technologies have led to the proliferation of various computational devices – from desktop machines to small appliances and portable devices - that are network-enabled (EC-ISU DG, 2006). This trend, which has brought increasing connectivity, collaboration, and access to various networked resources and services, has not only impacted the way we use computers, but also changes the way we create applications for them (Zang et al, 2004; Zang et al, 2007). Software applications that were previously developed as monolithic entities and ran as "silos" are now being developed using components distributed on a network such as the Internet. In the foreseeable future, most software applications will no longer be built from the scratch, but composed on-the-fly by reusing, as much as possible, existing collection of distributed resources and services available in an ecosystem (NGG Report, 2006). To this end, the current Internet infrastructure is transiting to a fully formed computing environment with capacity to offer services. *In the emerging era of so-called "Internet of Services* (IoS)*"[1], independently-developed distributed services will be utilized on-the-fly to fulfil a range of application needs (Weske et al, 2009).*

Service Oriented Computing (SOC)[2] has emerged as the key Software Engineering paradigm driving this change and facilitating value-creation from

[1] The term *"Internet of Services"*, also referred to as *"Service Grid"* in some literatures, is a vision of an Internet infrastructure where services become ubiquitous and the building block for composing software applications.

[2] The term *Service Oriented Computing (SOC)* is used in this thesis to describe services that are delivered purely by computational elements (i.e. software). In some literature, the term

existing assets. Its methodologies and abstractions are widely believed by major practitioners to provide the right specifications and standards for effective utilization of network-accessible services as part of a distributed application (Endrei et. al, 2004; Singh and Huhns, 2005). The specific challenges posed by SOC in this regard, as different from previous software engineering methodologies, are in how to *discover*, *select*, *compose* and *manage* services that are part of a distributed application (EC-ISU DG, 2006). These challenges become non-trivia when services need to function in open and dynamic environments where the service landscapes can be massive and such services can come and go at whim. Thus, when dealing with *service-based applications*, it is useful to consider how these applications are built and how services should function together within them (Huhns & Singh, 2005). Obviously, applications will use services by composing or putting them together. Hence, service composition becomes the *raison d'etre* for services because it lets us create new value from existing assets.

In addition, the growing importance of service composition to the vision of a global "Internet of Services" is generating renewed research interest in novel middleware technologies and techniques for supporting the vision of SOC (Issarny et al., 2007). The enactment of service composition demands a direct interaction with middleware to provide support infrastructure and runtime execution environment for services (Polze and Troger, 2008). Generally, the middleware form the base for the service-oriented distributed application (Sun and Blateky, 2004; Tanenbaum and Van Steen, 2007).

Almost a decade ago, Burbeck (2000) suggested that dynamic binding of services at runtime and automated composition of application from distributed services is the key to the future of service-based applications. But, according

---

Service Oriented Architecture (SOA) is also used synonymously (especially in business services). In most cases, SOA is a superset of SOC and has a broader meaning than capabilities delivered by computational elements as is used in this thesis.

to Weske (2008) and Woods et al (2009), this goal is still yet to be achieved. Although, it could be argued that there are a number of middleware platforms and tools for service composition that have been proposed and used in many scientific and industrial projects such as Triana (2003); Taverna (2004); myGrid (2004); and JIGSA (2005). Most of them are based on workflow techniques and realized only a minimal aspect of service composition (Issarny et al., 2007). The limitations of current state-of-the-art service composition technique include:

(i) **Lack of flexibility:** Service invocation orders are predetermined at design time, which means they are only constrained to the services whose specifications are available in the platform during design of the workflow. Their structure is, therefore, rigid because they are constructed prior to use and are enforced by some central authority. The need for flexibility is, however, apparent when exception occurs and rigid workflows behave incorrectly. Fault and exceptions are manually planned, whereas it is practically impossible to specify all exceptions statically and in advance (Singh and Huhns, 2005).

(ii) **Lack of Scalability:** Performance degradation due to lack of scalability is another limitation of service composition based on workflow. The majorities of workflow techniques are based on orchestration paradigm and, therefore, rely on a centralized coordinator. The recent trends in Service Oriented Computing is, however, towards pervasive services where services permeates our everyday life and is available in the home, offices and appliances we use (Huhns et al, 2005). Hence, the scale will be massive. In this sense, decentralization of control in the service delivery infrastructure is essential to improve scalability, reduce communications costs and avoid single locus of failure (Verma, 2004; Bhatia, 2005).

(iii) **Poor Adaptation Mechanism:** The current state-of-the-art service composition assumes a closed and static environment for services, whereas services operate in open and dynamic environments where user and system requirements are rarely static (Weske, 2008). A workflow's design context might not remain applicable in the workflow's execution lifetime. Dynamic environments often necessitate arbitrary extensions not recorded in the workflow model itself. Hence, the system cannot be dynamically modified when new user or system requirements are available at runtime.

(iv) **Poor Interactivity:** Interactions among services for effective composition is limited in current solutions because services are inherently not communicative. Services are by nature "passive" until they are invoked and they cannot react intelligently to changes in their execution environment (Singh and Huhns, 2005). Therefore, to achieve a higher level of interaction between services especially to facilitate meaningful negotiation and commitments requires a new approach to service interactions that is deeper than both orchestration and choreography.

The above outlined challenges of service composition in open and dynamic environment motivated this thesis. Specifically, the thesis investigates an answer to the question: *how can we effectively put services together as part of a distributed application in open and highly dynamic environments*? To answer this question, the thesis makes the case for a new service composition strategy that is *flexible* and *adaptive* to both user and system requirement changes, especially in highly dynamic and open environments. We believe that service composition should be planned automatically and executed transparently by the middleware infrastructure at runtime rather than relying on a workflow planned apriori by the human user. Besides, services must become "alive" and "active" in order to react flexibly and adapt to runtime

changes in its requirement. We think such a new strategy would lead to a scalable, fault-tolerant, and cost-effective service composition process.

To elaborate our solution approach, we present MINDS: a *M*iddleware *IN*frastructure for *D*istributed *S*ervices provisioning formulated in this thesis (Iyilade et al, 2009). In the first instance, MINDS design and architecture are structured around autonomous "active" services which are "aware" of their operating environment and thus, can sense "new opportunities" and react flexibly to runtime failure and requirement changes. Besides, we also propose a new life-cycle process for service provisioning in dynamic environment based on MINDS. Unlike the current state-of-the-art service provisioning process that requires manual specification of data and control flow by the human user at design time, MINDS service provisioning process realizes runtime, automated composition optimized for a particular user request on-demand. Furthermore, to demonstrate the utility and applicability of MINDS concepts in a real service composition scenario, we implemented an experimental prototype for the case of a collaborative e-Tourism Virtual Enterprises. Finally, to evaluate MINDS performance in a repeatable and controlled manner, we also carry out empirical analysis and simulation experiments to compare MINDS performance with alternative design options and related strategies found in the literature.

The rest of this chapter is organized as follows: Section 1.2 presents the thesis goal while section 1.3 outlines the objectives of the thesis. Our Research Methodology is presented in Section 1.4, while Section 1.5 discusses the Research Contributions of the Thesis. The chapter ends with Section 1.6 with arrangement of the rest of this thesis.

1.2 Thesis Goal

The main goal of this thesis is to investigate how to effectively compose services in a highly dynamic and open environment. More specifically, the thesis investigated the research question: *how can we effectively put services together as part of a distributed application in open and highly dynamic environments*?

1.3 Thesis Objectives

Based on our research goal towards investigating how to compose services in a highly dynamic and open environment, the following objectives were set towards attaining this goal:

(i) to formulate a flexible and adaptive middleware architecture for distributed service provisioning in open and dynamic environments;

(ii) to demonstrate the utility and applicability of the middleware architecture proposed in (i) above on an appropriate test-bed;

(iii) to evaluate the performance of the middleware using metrics such as *scalability*, *fault-tolerance*, and *network bandwidth cost optimization*.

1.4 Research Methodology

The following methodologies were employed in fulfilling the above stated objectives:

(1) *Literature Survey:* The aim of the literature survey was to establish a good theoretical foundation for the research. We started by exploring, from the literature, some recent trends in the area of Service Oriented Computing with special focus on how they support service composition in open and dynamic settings. We also reviewed other software engineering methodologies such as Agent Oriented Software Engineering (AOSE) and the Semantic Web to see how their methodology and paradigm enable a flexible and adaptive system. Thereafter, a comprehensive survey of some existing middleware platforms for service composition was carried out. We analyzed existing service composition strategies based on the following criteria: *service discovery, binding, and selection method used; how process*

models are created; the execution models used; and finally, *the architectural and design decisions they make;*

(2) *Model Formulation:* Based on a concrete real-life sample scenario and the insights gained from literature on recent trends in the areas of Service Oriented Computing, AOSE and the Semantic Web, a set of design criteria that a flexible and adaptive middleware should support were drawn. This was then transformed into a middleware architectural model and design for dynamic service composition.

(3) *Prototyping:* An experimental prototype implementation of the middleware was carried out using Java programming language and some other relevant open-source frameworks in the fields of Service Oriented Computing and Agent Oriented Software Engineering. Prototyping provides us a mechanism to elaborate and demonstrate the utility of our concepts in a real-life setting.

(4) *Evaluation:* To show the reliability of our service composition strategy, its performance needs to be evaluated. Testing a service composition system on a really global service infrastructure is a challenge because no single user has control on the entire system. In this regard, we rely on an empirical analysis and simulation experiments to test the performance of our service composition strategy against related work in the literature. Our aim was to investigate how the design choices of each of the systems led to performance gain/loss in terms of *scalability*, *fault tolerance*; and *network bandwidth cost optimization*. First, a mathematical model was developed for the systems. Second, through simulation, we carried out various experiments that investigated the following:

   a. the effect of increasing number of registered services on service discovery time.
   b. how increasing the number of tasks in the composition affects service composition time.

c. how the failure of service provider(s) affects the number of composition tasks that met deadline.
d. how the failure of the service monitoring impacts the number of composition tasks that met deadline.
e. the effect of increasing the number of interacting service providers on the bandwidth cost.

1.5 Thesis Contributions

Dynamic service composition remains an open challenge today in Service Oriented Computing (Weske, 2008; Woods, 2009). Achieving runtime binding to services is a key to the future success of Internet-based services since services operate in open environment where user and system requirement changes are the norm (Singh and Huhns, 2005). In this thesis, we argue that in order to support these changing user and system requirement, we need a flexible and adaptive service composition strategy where service composition is planned and executed automatically by the middleware infrastructure at runtime rather than relying on a statically defined workflow by the human user as found in many existing systems. This strategy led to a more efficient service composition system in terms of scalability, fault tolerance and bandwidth cost optimization.

To support this thesis and elaborate our solution approach, we have designed and developed **MINDS** – a **M**iddleware **I**nfrastructure for **D**istributed **S**ervice Provisioning as a platform to analyze our service composition strategy. Through MINDS, we have made the following key research contributions:

(1) **the thesis presents a new middleware architecture based on decentralized "active" services:** Due to the importance of service composition in SOC, many middleware systems for service composition are already in use and found in many industrial and scientific projects such as Triana (2003); Taverna (2004); myGrid (2004); and JIGSA (2005). However, most of these systems are based on procedural techniques where service composition is viewed as a workflow and rely on a central orchestration engine for composition enactment. This makes them rigid and suffer scalability problem when applied in dynamic and open settings where requirement for composition may change at runtime and

service landscape can be massive (Singh and Huhns, 2005, Bhatia, 2005). To address the performance bottleneck involved in a centralized workflow system, Barker et al (2009b) suggested the need for a *decentralized architecture*. They then propose an architecture that is based on *service choreography.* However, while choreography is based on simple message passing without the need for a central orchestrating engine of a normal workflow, the interactions and communication among services is still limited because services are naturally "passive" until invoked and, therefore, unaware of changes in their execution environment (Singh and Huhns, 2005). To address the above challenges, we present a new flexible and adaptive middleware architecture for distributed service provisioning, called MINDS.

MINDS architecture is structured around autonomous "active" grid/web services that are able to proactively take runtime composition decisions based on current situation and evolving circumstances. The grid/web services were made active by wrapping them in software agents. Hence, the system operates through a goal-based *collaborative conversation among software agents* thereby leading to flexible collaboration and richer interactions among services. In addition, MINDS overcome the performance bottleneck of a centralized architecture found in many workflow systems by utilizing a decentralized team of software agents for matchmaking and composition. Chapter 3 of this thesis elaborates on this contribution in details.

(2) **the thesis presents a Service Provisioning Life-Cycle process for automated, runtime composition:** Service composition based on automated, runtime composition has been identified as the key to the future of SOC (Burbeck, 2000). The life-cycle process involved in provisioning services as part of a distributed application involves service discovery, selection, composition and binding (Huhns and Singh, 2005). Some or all of these steps could be performed at *design time* or at *runtime*. In addition, some of the steps could be carried out manually by human, or they may be semi- or fully automated by the middleware system (Fluegge *et al*, 2006). Currently, many service composition platforms in use today are based on workflow and adopt a service provisioning process that is based on *manual, design time composition* (Kowalkiewicz, 2008;

Fluegge *et al*, 2006; Alamri et al, 2006). However, this is undesirable in dynamic settings for a number of reasons: First, service composition may become inadequate or even incorrect during runtime execution because of a change in user and business requirements thus, necessitating changes in the composition. Second, it can be error-prone and complex since it requires the engagement of a highly qualified specialist who needs to understand the composition requirement and what available services do. Third, it can lead to in-optimality since service compositions are not tailored to individual service request.

Shifting service composition from manual to automated and from design time to runtime is the solution to these challenges (Kowalkiewicz, 2008). We, therefore, propose a new life cycle process that achieved runtime, automated compositions based on the MINDS architecture. Unlike conventional workflow life-cycle process, which requires a manually defined control and data flow by the human user, MINDS service provisioning process is based on just preconditions and goal of the composition as specified by the user request. The service provisioning life-cycle has as input user request and as output the result of the composition. It is achieved as a four-step process involving: *task decomposition and representation*; *dynamic service matchmaking*; *composition synthesis*; and *execution monitoring*. Chapter 4 discusses this contribution in detail.

(3) **the thesis demonstrates the effectiveness of key strategies for service composition:** We present a comprehensive performance analysis and simulation experiments to investigate various service composition strategies and design options using our own strategies in MINDS as a benchmark. Our empirical analysis and performance results offer a useful guide for practitioners in the trade-offs and merits of various service composition systems' design strategies. Especially, those that need to possess the three crucial performance qualities of *scalability*, *fault-tolerance* and *bandwidth cost-effectiveness.* Chapter 6 of this thesis elaborates this contribution in detail.

1.6 Thesis Outline

The remainder of this thesis is structured as follows:

In chapter 2 we provide a background to the problem addressed in this thesis and present the justification for a new service composition strategy and system. We then provide a brief overview of our service composition strategy and highlight our contributions.

Chapter 3 presents the conceptual design and architecture of MINDS, a middleware infrastructure for distributed services provisioning that was formulated in this thesis. We describe the context of MINDS and the design decisions we made. Thereafter we present the architecture and describe its core functionalities.

In Chapter 4, we discuss a new life-cycle process for service composition. The chapter elaborates how to achieve flexibility and adaptation to user and system changes through an automated on-demand service composition strategy that is customized to individual requests and does not require the involvement of a human modeler.

Next, in Chapter 5, we discuss an experimental prototype implementation of MINDS, which provides a test-bed for demonstrating the utility and applicability of MINDS in a real-life scenario.

In Chapter 6, we focus our attention on evaluating the performance of MINDS using an empirical analytical approach. First, the chapter presents a mathematical analysis that compares MINDS' design decisions with some related work found in the literature. Second, the chapter also presents the results of extensive simulation experiments carried out based on this analysis.

We conclude the thesis in chapter 7 with a summary of our research work, its limitations and an outlook on future work.

This is finally followed by the appendices which provide further information on publications from this thesis and sample source code listing.

Chapter 2

Background

In this thesis, we propose an approach that overcomes the limitations of current service composition strategy that are based on workflow. First, our service composition platform, called MINDS (Middleware Infrastructure for Distributed Services Provisioning), is based on the model of business processes as a collaborative conversation among software agents which, therefore, allows us to capture runtime behaviors that can impact the composition that were not planned apriori. Second, our strategy enables late (runtime) binding to services, unlike the design time binding in most service composition workflow. Third, our approach also allows richer interactivity of services by enabling services to be active and aware of changes in the user and/or execution environment which then lead us to a *flexible and adaptive service composition strategy*. This chapter identifies the challenges of service composition in open and dynamic environments through a case study scenario of e-Tourism virtual enterprise collaboration. It then presents an overview of current-state-of-the-art service composition strategies and highlights their shortcomings and limitations. We introduce our own strategy towards addressing these shortcomings and overview its key features.

2.1 Introduction

At the heart of any distributed service provisioning infrastructure is the composition of services (Weske, 2008). Service composition has gained considerable attention recently as a means of developing distributed application than previous techniques that are based on top-down decomposition (EC-ISU, 2006). Hence, software could no longer be developed from scratch as monolithic entities as in the past, but rather composed by reusing as much as possible

existing assets and components available on the network (EC-ISU DG, 2006; Singh & Huhns, 2005; Nessi-Grid, 2006). *Service Oriented Computing (SOC)* paradigm is currently widely seen as an effective methodology towards realizing the goal of developing distributed applications through composition.

To develop *service-based applications*[3], various atomic services are often engaged by composing or putting them together. It, therefore, becomes important to examine how these applications are built and how services should function together within them (Singh & Huhns, 2005).

Service composition in open and dynamic environment such as the Internet is still a major research challenge for the SOC community today. The vision of dynamic binding of services at runtime, which has been identified by Burbeck (2000) as a core of the future of service oriented environment, is still yet to be achieved (Woods et. al, 2009; Weske, 2008). Currently, most service infrastructures (grid/web services) adopt service composition techniques that are procedural and based on workflow. Infact, today, there are over 100 workflow management systems in use with the most popular ones seen in projects such as Triana (2003), Taverna (2004), and JIGSA (2005). Workflow techniques generally follow a predefined order of service execution and binding prior to its enactment and are, therefore, too rigid and inflexible to user and system requirement changes that are not planned at design time.

Contrary to the assumption of a closed, static setting in most conventional service composition techniques, services often operate in open and dynamic environments. For example, most business organizations require the services of an external party or provider such as a financial entity, delivery service, etc, to meet the need of their customer. Therefore, it is useful to think in terms of such open environment when

[3] *Service-based application* are composed by utilizing services available locally within an enterprise or distributed on the network.

dealing with services and service composition (Singh and Huhns, 2005). From the foregoing, it is obvious that this limitation of current state-of-the-art service composition strategy needs to be addressed through a service composition strategy that is flexible and adaptable to runtime changes in user and system requirements.

This chapter motivates the specific problem that this thesis addressed and presents an outline of our approach. The rest of the chapter is structured thus: Section 2.2 presents an example scenario that highlights the challenges of service composition and motivates the need for a flexible and adaptive middleware for service composition. Thereafter, Section 2.3 presents current state of the art strategy for service composition, while Section 2.4 presents our own strategy based on a flexible and adaptive service composition system. We discuss the key distinguishing features of our work. Finally, Section 2.5 concludes the Chapter.

2.2 Motivating Scenario: e-Tourism Collaborative Virtual Organization

The challenges of service composition in dynamic and open environments can best be understood through an example scenario (Adigun et al, 2009). In this section we introduce a case study of e-Tourism collaborative virtual organization that will be used in this chapter and some other chapters in illustrating the problems of service composition that we address in this thesis. The scenario provides a coherent, comprehensive, and internally consistent description of challenges of service composition in dynamic environments. However, it is important to keep in mind that this is just one specific example; the service composition problems addressed in this thesis are independent of any specific domain.

We assume a South African-based travel agent company, called **Planit (Pty) Ltd** (*hereafter, referred to as* **Planit**). *Planit* specializes in planning trips of her clients which, mainly, include **flight booking**, **hotel booking** and **car rental**. It offers a web/mobile interface that allows clients to make travel arrangements and also specify their trip preferences.

Let us consider a South African businessman, named, **Joe**, who intends to use **Planit** travel service to arrange his next business trip from ***Durban*** to ***Johannesburg***. **Joe** will access **Planit**'s web/mobile interface to make travel arrangements for his trip. In this case, **Joe** is interested in the following:

(i) booking a flight from Durban to Johannesburg that would not cost him more than R1,500;

(ii) renting an executive-class car that he would drive from the airport to the place of his business meeting;

(iii) booking a hotel that is not far from the location of his business meeting.

Accessing the **Planit** web interface on his browser, **Joe** is able to specify all the information required for reservation processing. These include, *the time and day of his trip*, the *type of room he needs* and in *what location*, *payment details* and so on.

As illustrated in Figure 2.1, to process **Joe**'s request, **Planit** web application *Order Receiver (OR)* will access the company's internal services such as the *Customer Relations Management (CRM)* as well as external partner services such as *Airlines*, *Hotels*, *Car Rentals* services since **Planit** is member of a collaborative network of tourism businesses in South Africa, who formed a virtual organization for the purpose of sharing their business services. The order processing logic may also use *Financial Institutions* such as Banks for all payment activities.

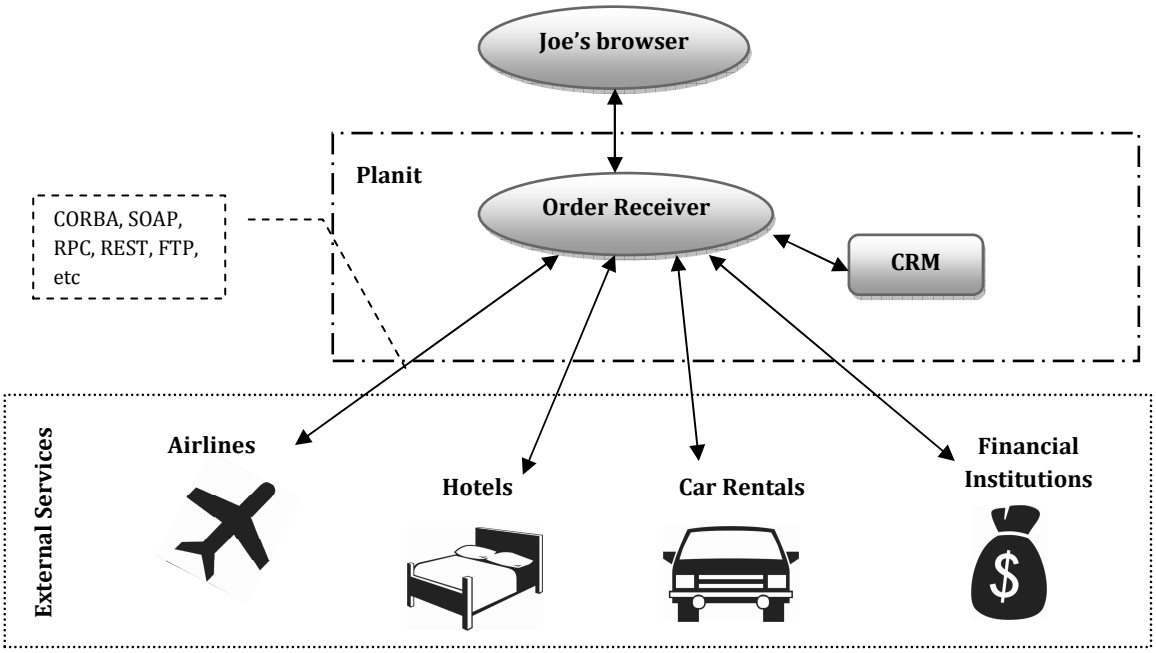

Figure 2.1: Illustrating the Planit relationship with internal and external business services

To take advantage of the business services available in the virtual organization, **Planit** has to expand its current point-to-point communication with the various internal and external services. *Planit order receiver* now connects with a service delivery *middleware platform* that integrates all the business entities in the South Africa Tourism collaboration. The goal of the middleware platform is to provide a Business-to-Business (B2B) solution that reuses business capabilities of the various entities in the virtual organization.

Figure 2.2 illustrates how Planit system architecture was expanded to support B2B collaboration and access to services using a central middleware platform. The benefit of unifying and channeling access to the company's internal and external services is an increased flexibility and maintainability. For example, to integrate a new service, it is first published in the middleware platform and then access control rights can now be centrally managed.

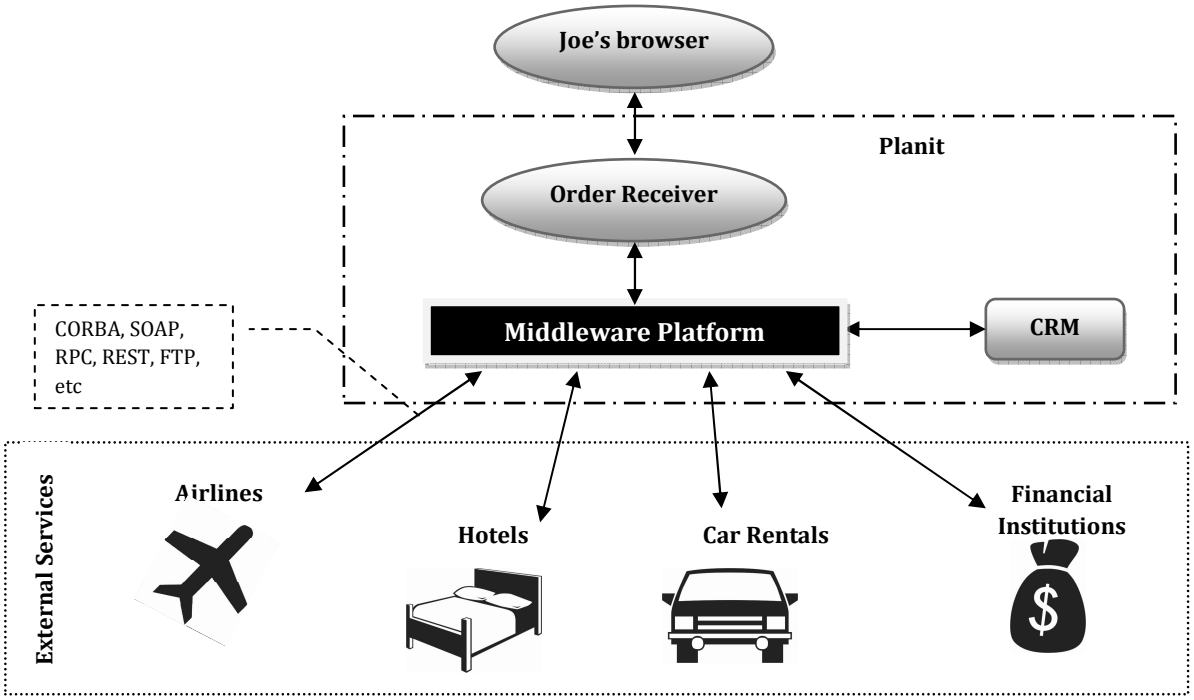

Figure 2.2: Connecting all collaborating services through a unified middleware platform

2.3 Current State-of-the-Art Service Composition Strategy

2.3.1 Service Composition as a Workflow

Service composition involves extended, loosely coupled activities among two or more autonomous business organization. Such activities can be thought of as (business) process that engages several services in a manner that brings about the (desired) outcome.

Many researchers today model processes as a ***workflow*** (Singh and Huhns, 2005). And, although, there is a fine line between processes and workflows, some research seems to treat them as isomorphic. Infact, most of the researches on processes are based on previous research on workflow. For example, one of the inputs to the current leading standard for processes, Business Process Execution Language for Web Services

(BPEL4WS) (Curbera et al, 2002) is the Web Services Flow Language (WSFL) by IBM (Snell, 2001).

As defined by the Workflow Management Coalitions (WFMC, 1995):

> *A workflow is the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant (a resource, human or machine) to another for action, according to a set of procedural rules).*

Hence, a workflow is an activity that addresses some business needs by carrying out specified control and data flows among sub-activities.

Workflows are described using workflow definition languages, which can be categorized into *web service-* and *grid service- oriented* languages. Web service-oriented languages include Microsoft XLANG (Thatte, 2001), Web Service Flow Language (WSFL) (Snell, 2001), Web Service Choreography Interface (WSCI) (W3C, 2002a), Business Process Execution Language for Web Services (BPEL4WS) (Curbera et al, 2002), and Business Process Management Language (BPML) (BPML, 2002). Grid service-oriented languages are built on top of their web service predecessors with special requirements for dynamic settings. The grid service category includes workflow languages such as Grid Workflow Execution Language (GWEL) (Cybok, 2004), Grid Service Flow Language (GSFL) (Krishnan, et al, 2002), GALE (Beiriger et al, 2000), and SWFL (Huang & Walker, 2003).

A workflow can be described from the viewpoint of a single participant using *orchestration* or from a global perspective using *choreography* (Singh & Huhns, 2005; Barker et al, 2009b).

Orchestration enables services to be composed together in predefined patterns, described using an *orchestration language* such as *Business Process Execution Language for Web Services (BPEL4WS)* and executed on

an *orchestration engine* such as Oracle BPEL Process Manager[4] and Active BPEL Engine[5].

All interactions that are part of a business process (including the sequence of activities, conditional events, amongst others) must be described. This description is then executed by a central orchestration engine, which has control over the overall composition. This is done as per the requirements of the orchestration (Juric et al, 2006).

The involved services do not know they are involved in composition and are part of a higher business process. Only the central coordinator of the orchestration knows this. So, the orchestration is centralized with explicit definitions of operations and the order of invocation of services. For example, as shown in Figure 2.3, Services 1, 2, 3…n do not interact directly as part of an external business process. Their interaction is through the central orchestration (coordinator) engine.

Choreography, on the other hand, does not rely on a central coordinator but rather takes the view that a composition is a set of message exchanges between participants. As illustrated in Figure 2.4, it is focused on exchange of messages in public business processes. The message exchanges are constrained to occur in various sequences and may be grouped into various transactions. Hence Service 1, 2, 3, and 4 can exchange messages directly without going through a central orchestration engine as was the case in Figure 2.3. The distinction between orchestration and choreography can be made as follows: *Orchestration can be compared to a set of musicians (services) commanded by a conductor (engine), while choreography can be compared to a group of dancers (services) that already know how to perform and that do not obey a central coordinator.*

[4] Oracle BPEL Process Manager – http://www.oracle.com/technology/products/ias/bpel/index.html
[5] Active BPEL Engine – http://www.activebpel.org

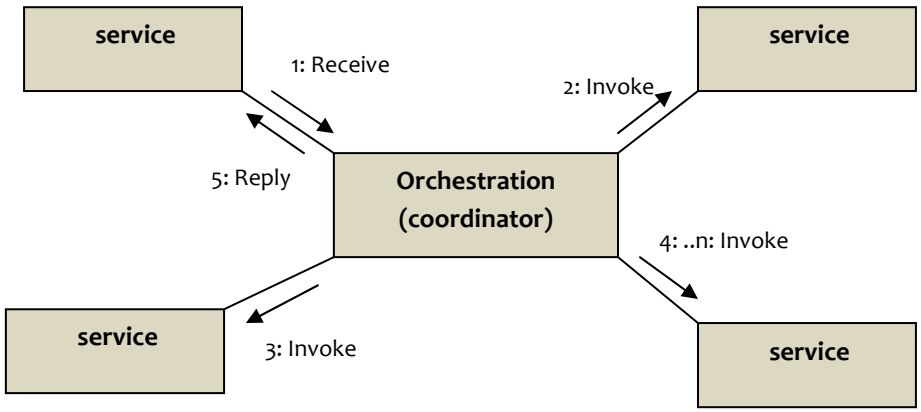

Figure 2.3. Workflow description using orchestration (*adapted from Juric et. al, 2006*)

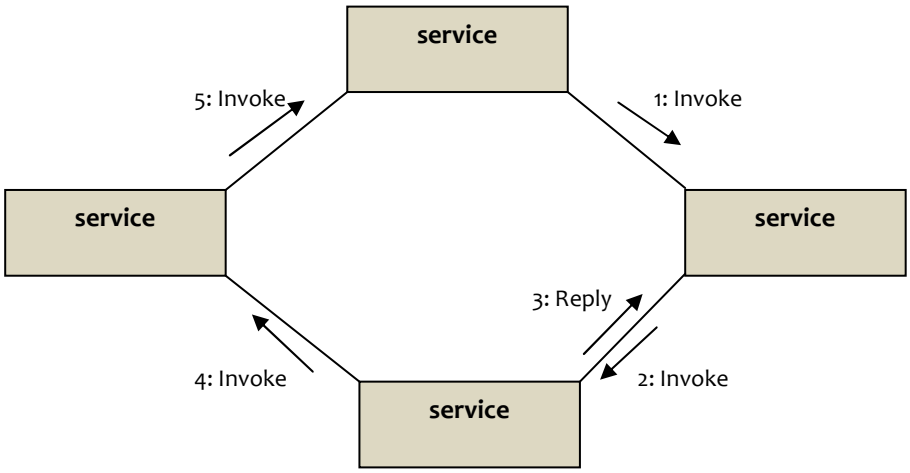

Figure 2.4. Workflow description using Choreography (*adapted from Juric et. al, 2006*)

If we look at the example given earlier, using current state-of-the-art service composition strategy, Planit order receiver will make use of a predefined ***workflow*** to aggregate the required services in the platform that comprise the travel package. This is illustrated in Figure 2.5.

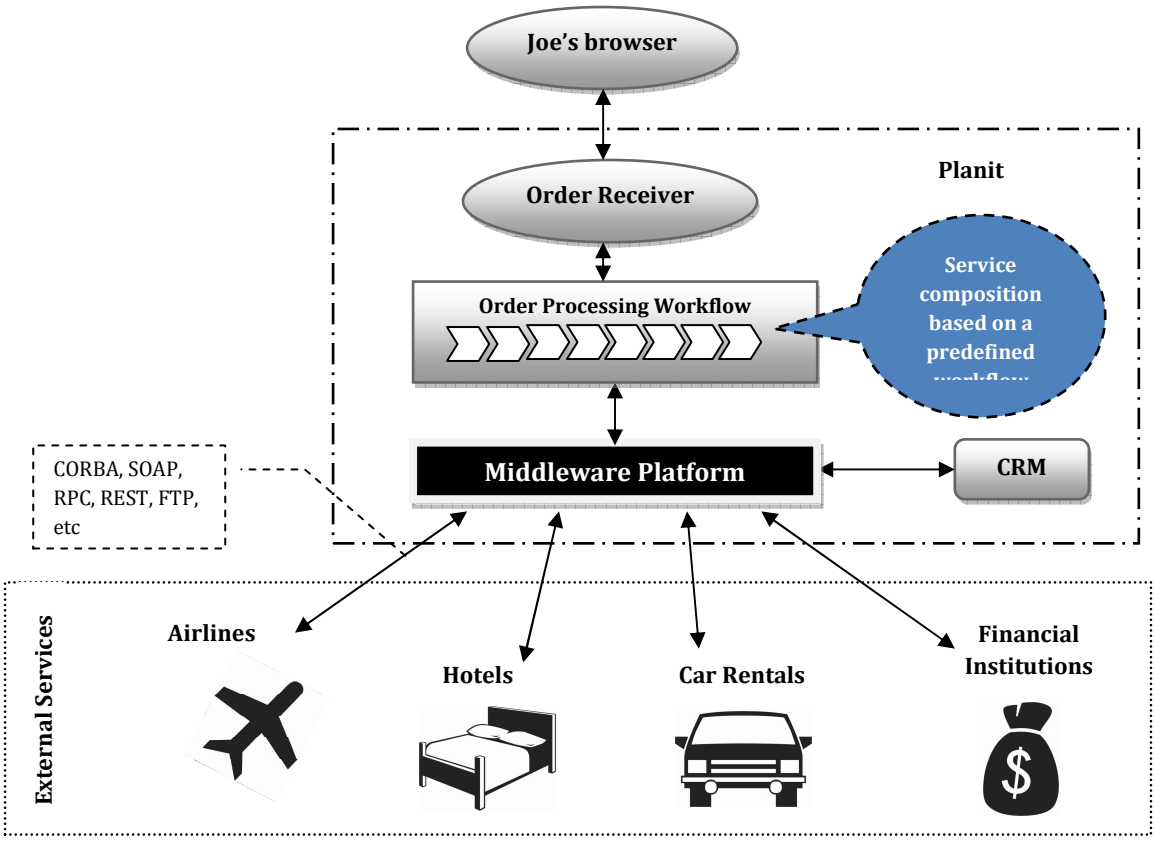

Figure 2.5: A predefined order processing workflow used to aggregate services in the platform required to process Joe's order

2.3.2 Drawbacks of Service Composition Strategy based on Workflow

Workflow technique is currently the prevalent strategy for service composition. Infact, there are more than one hundred workflow management systems in use today, but the most popular ones are seen in many industrial and scientific workflow projects such as Triana (Triana, 2003), Taverna (2004), myGrid (2004), and JIGSA (2005).

Using the current state-of-the-art service composition middleware platform where services are composed as a workflow has a number of drawbacks:

(1) ***Lack of Flexibility:*** In a workflow, service invocation order are predetermined at design time, which means they are only constrained to the services whose specifications are available in the platform during design of the workflow. The workflow structure is, therefore, rigid because they are constructed prior to use and are enforced by some central authority. Additionally, the rigidity of workflow also causes productivity losses by making it harder to accommodate the flexible, *ad hoc* reasoning that is strong suite of human intelligence (Singh and Huhns, 2005).

The need for flexibility is actually most apparent when exception occurs and rigid workflow management tools behave incorrectly. For example, assuming in checking whether Joe already has an account with Planit for charging, the system discovers some unpaid and overdue balance – or that someone else previously with the same address has an unpaid balance. Such discoveries would raise a red flag that might lead to a modification of the workflow or a cancellation of the whole order. Even though, exceptions and fault handling capabilities are manually planned in many workflow tools, it is practically impossible to specify all exceptions statically and in advance. As a consequence, many of the workflow processes become inflexible.

(2) ***Lack of Scalability:*** Performance degradation due to lack of scalability is another limitation of service composition based on workflow. Although, a workflow can be described as a choreography, which is a decentralized architecture, in reality, majority of workflow research and projects in both e-Commerce and e-Science are based on orchestration paradigm and, therefore, relies on a centralized coordinator. Centralization brings scalability problem as the number of user requests and services available in the platform increases. Even though, the travel planning service

offered by Planit, as described in this scenario, deals with only a handful of atomic services, there are times when the scale of atomic services becomes so large that the execution of the composite service takes a very long time to complete. The recent trends in service oriented computing is towards pervasive services where services permeates our everyday live and is available in the home, offices and appliances we use (Huhns et al, 2005). Hence, the scale will be massive. In this sense, decentralization of control in the service delivery infrastructure is essential to improve scalability, reduce communications costs and avoid single locus of failure (Bhatia, 2005).

(3) ***Poor Adaptation Mechanism:*** Another notable challenge with service composition as a workflow is poor adaptation mechanisms. User and system requirements are rarely static. A workflow's design context might not remain applicable in every detail over the workflow's lifetime. Dynamic environments can necessitate arbitrary extensions not recorded in the workflow model itself. Hence, the system cannot be dynamically modified when new user or system requirements are available at runtime.

In the earlier described scenario, for example, if Joe wishes to add another request to check the weather conditions of his destination, it becomes practically impossible to dynamically incorporate a weather service to meet Joe's demands. What about accommodating some other services that may make Joe's trip pleasurable that were not even known to the user or Planit Order Receiver apriori? How about having a flight monitoring service that checks and notifies of flight delays, can it participate in Joe's itinerary plan automatically at runtime based on some semantic awareness of the value it could add to the itinerary planner?

In this sense, it is clear, that the dynamics of service composition as illustrated in this case study would defy a predefined sequence of actions or workflow. Therefore, a more flexible and adaptive strategy becomes essential.

(4) ***Poor Interactivity:*** Even though, choreography allows some form of interaction between cooperating services through passing of public messages (Barker, 2009a), however, interactions among services are limited because services are inherently not communicative. Services are by nature "passive" until they are invoked and they cannot react intelligently to changes in their execution environment. Therefore, to achieve a higher level of interaction between services especially to facilitate meaningful negotiation and commitments requires a new approach to service interactions that is deeper than both orchestration and choreography. Accordingly, we think such deeper interactions can be facilitated by complementing current service technologies with some other Software Engineering methodology such as Agent Oriented Software Engineering hence, service composition can simply be modeled as *collaborative conversation among teams of software agents*.

2.4 A Flexible and Adaptive Service Composition Strategy

Current state-of-the-art approach where service composition is treated as a workflow orchestrated by a central execution engine has a number of drawbacks as outlined in Section 2.3.2. During design time, the application developer selects the services that would participate in the composition and statically bind them to the applications. That means the system is inflexible and not adaptable to runtime changes of the user or service environment because it is hard-wired to the services in the platform. This, in our opinion, results in poor utilization of services

especially because it becomes harder to incorporate new, cheaper or better services that become available on the middleware platform seamlessly.

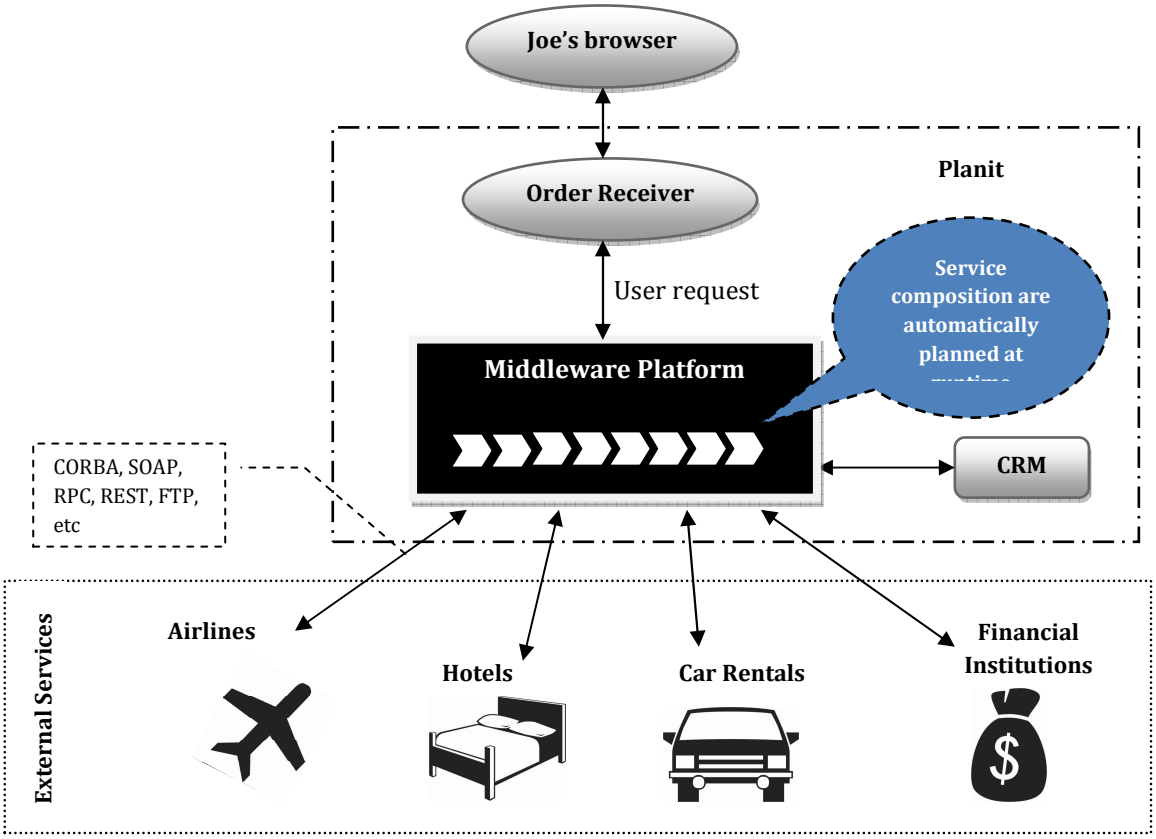

Figure 2.6: High-level overview of a flexible and adaptive Service Composition Strategy

In this thesis, we propose an approach that overcomes the limitations of current service composition strategy that are based on workflow. First, our service composition platform, called MINDS (Middleware Infrastructure for Distributed Services Provisioning), is based on the model of business processes as a collaborative conversation among software agents[6] which therefore allows us to capture runtime behaviours that can impact the composition that were not planned apriori. Second, our strategy enables late (runtime) binding to services unlike the design time binding in most service composition workflow. Third, our

[6] The definition of the term *agent* is often debated in the literature. However, throughout this thesis, a software agent is defined as a computational entity that has persistent identity; can perceive, reason about, and initiate activities in its environment; can communicate with other agents, including human.

approach also allows richer interactivity of services by enabling services to be active and aware of changes in the user and/or execution environment which then lead us to a *flexible and adaptive service composition strategy*.

If we go back to our earlier scenario example, the high-level overview of the new flexible and adaptive service composition strategy is illustrated in Figure 2.6.

Here, unlike in the conventional (workflow) strategy where the services to bind to are predefined, the input to the middleware platform does not include a description of what shall be achieved and which concrete service has to be executed during the composition. A client request is rather a specification of *the task to be performed* and *the constraints* (such as, the quality of service (QoS) preferences of the end-user, e.g price, deadline, etc) associated with it.

We now briefly highlight the essential features of our service composition strategy:

2.4.1 Matchmaking based on Distributed "Active" Services

Service matchmaking is the first phase in any service provisioning. It is the process of finding suitable services for a task in the service composition. It, basically, involves two steps: *service discovery* and *service selection*.

The industry standards available to register and discover services are based on the *Universal Description Discovery and Integration* specification (UDDI 2002). Unfortunately, discovering Web services using UDDI is relatively inefficient since the specification does not take into account the semantics of Web services, even though it provides an interface for keyword and taxonomy based searching (Cardoso and Sheth, 2006).

It is now widely accepted that an effective means for the discovery of Web/Grid services than keyword or attribute-based matching is having semantics in the description of services itself (Berners-Lee et al, 2001; Sheth and Meersman 2002) and then use semantic matching algorithms to

find Web services (Smeaton and Quigley 1996; Klein and Bernstein 2001; Rodriguez and Egenhofer 2002; Cardoso and Sheth 2003).

Adding semantic annotations to Web Service Description Language (WSDL) specification and UDDI registries allows improving the discovery of Web services (Akkiraju et al, 2006; De Roure et al, 2001). The general algorithm for semantic Web service discovery requires the users to enter Web service requirements as templates constructed using ontological concepts. The algorithm matches services based on the functionality (the functionality is specified using ontological concepts that map to WSDL operations) they provide (Cardoso and Sheth 2003).

Besides annotating services with their semantic description, our technique for service discovery utilizes a set of distributed "active" grid/web services that pro-actively listens to an interaction space for any available published task that matches their semantic service specification. We have enabled services to be "alive" and "active" by wrapping them with Software Agents. The distinction between an *active* and *passive* approach to service discovery, as illustrated in Figure 2.7, is that when the discovery is *passive*, services have to wait until they are invoked by a *service broker* or *consumer,* whereas in an active approach the services can pro-actively listen and discover a matching task they can participate in the interaction space (e.g tuple space).

A very visible benefit of making services active through agents is that it encourages *"opportunity sensing"* whereby services are able to proactively discover any user task or composition request that are of interest to them and which match their semantic specification. This kind of requirement will be vital as services landscape become populous e.g in sensor networks, environmental monitoring, disaster management, where cooperating services will be required to dynamically organize themselves to achieve a particular composition goal without human involvement.

Additionally, conventional service discovery is often performed through a service broker that is centralized which then results in lack of scalability as the number of services in the ecosystem increases. We have employed decentralized service agents in order to facilitate efficient discovery and reduce the discovery and selection time of services when compared to conventional centralized matchmaker.

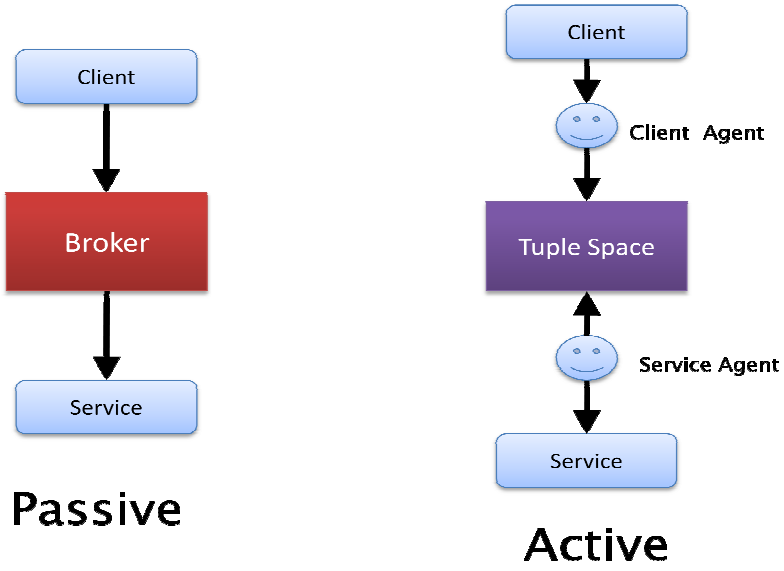

Figure 2.7: Illustrating the difference between passive and active approaches to service discovery

Another unique feature of our matchmaking strategy is in how services are selected after discovery. According to Singh (2004), the deeper challenge during matchmaking is not discovery which is the focus of UDDI and other efforts to annotate UDDI registries with semantics. The bigger challenge is that of *selection*. Unlike in many conventional techniques, such as those described in Cardoso and Seth (2003), where the outcomes of semantic service discovery are ranked and then the topmost are selected based on the ranking, we feel that selecting services just because it matches the semantic description of the user request violates the autonomy of the service provider. In certain instances, a particular service may not be interested in participating in a task even if it matches its semantic description because of a local policy or due to personal choice. In

view of these, we see the need to protect the autonomy of service providers by allowing a bidding process so that service agents whose semantic service specification matches the task to be executed autonomously bid for such task and be assigned the task if found competent. We employed the *Contract Net Protocol (CNP)* (Smith and Davies, 1981) in the bidding process. The CNP is a widely used interaction protocol for cooperative problem solving among software agents. It is modeled based on the contracting mechanism used by businesses to govern the exchange of goods and services. The CNP provides solution for the so-called *connection problem:* that is, finding an appropriate agent to work on a given task. The basic steps in the CNP protocol as shown in Figure 2.8 are:

1. a manager announces the existence of a task via a (possibly selective) multicast
2. agents evaluate the announcement. Some of these agents submit bids;
3. the manager awards a contract to the most appropriate agent

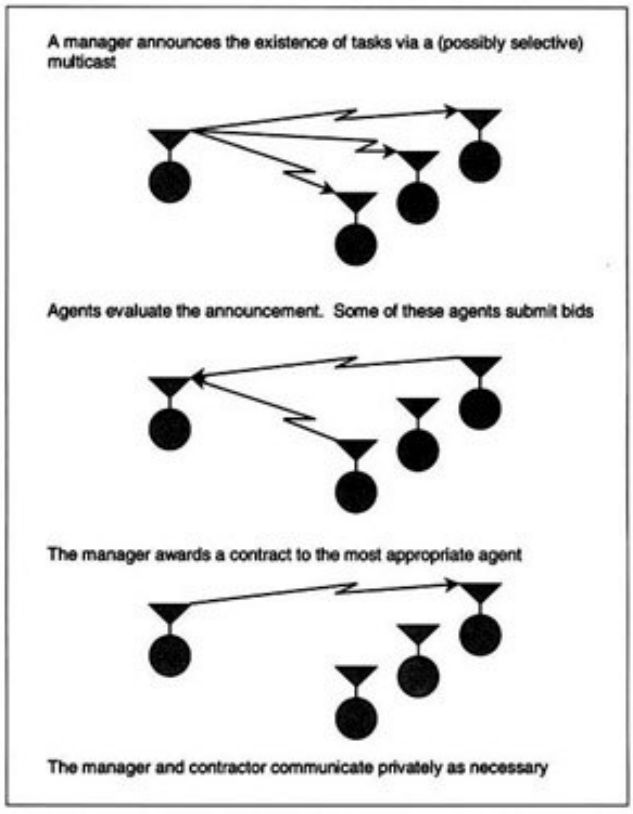

Figure 2.8: The Basic steps in the Contract Net Protocol (Weiss G ,1999)

Figure 2.9 illustrates the five basic steps involved in our service matchmaking strategy which includes the following:

(i) client requests are represented as composition tasks;

(ii) the tasks are published in the interaction space;

(iii) service agents listen for published tasks that matches their semantic service specifications;

(iv) matching services are discovered and they then bid to render the service, and

(v) the best bidding provider agent is selected and assigned the task.

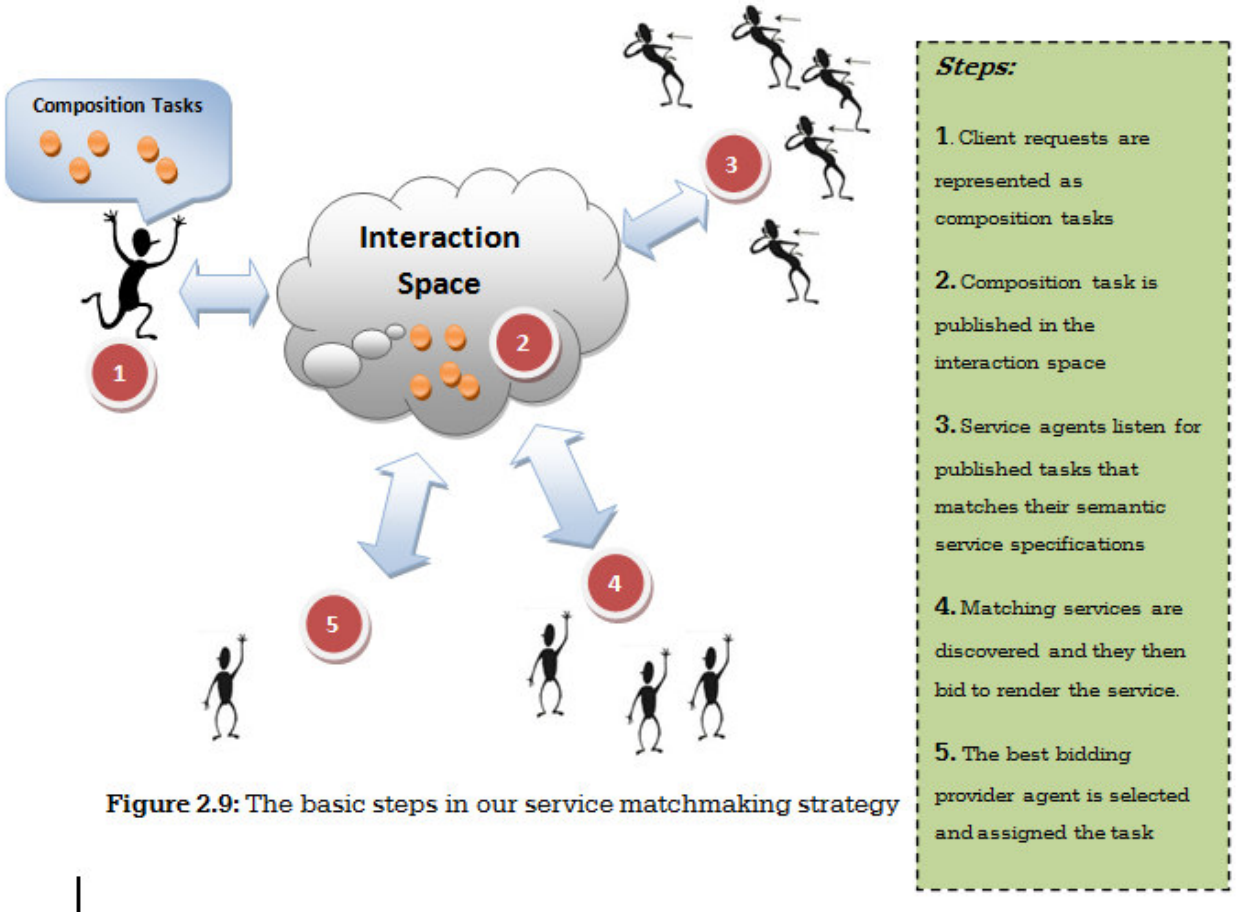

Figure 2.9: The basic steps in our service matchmaking strategy

2.4.2 Runtime, Automated Service Composition

The selection of the services which will participate in a service composition process may be done either at *design time* or at *run-time*. When done, at design time, the bindings are said to be *static* or *early* i.e. each instantiation of the composite service will be made up of the same constituent services (Alamri *et al*, 2006; Fluegge *et al*, 2006). But, when it is done at run-time, the composing services are selected based on automatically analyzable criteria such as service functionality, signature and quality of service (QoS) parameters. This is often referred to as *late binding*.

Our service composition strategy is based on late (runtime) binding to services. Applying late binding to services has some benefits. For example, in a growing service market, third party service providers may offer the same functionality under different conditions e.g QoS parameters like *price*. Therefore, late binding to services guarantee scalability as the number of services increase because the cost of composite service offered by a provider may decrease with growing competition in the associated marketplace (Fluegge *et al*, 2006).

Furthermore, late binding enhance fault-tolerance and thus reliability. Since the actions in a process are not hardwired to concrete services, the unavailability of a service may be compensated through the invocation of a semantically equivalent one. In addition, there are some scenarios where important service characteristics (like price) change constantly, which makes the use of run-time binding almost essential for the success of the composition.

Another very important feature of our service composition strategy is the *degree of automation in the creation of the process model*. As stated earlier, traditional service composition methods based on workflow require the user to define the data flow and the control flow of a composite service *manually,* either directly or by means of designer tools e.g Business Process Modeling Languages, BPML (BPML, 2002). Subsequently, the process description is deployed in a process execution engine. Depending on the abstraction level provided by the tools and also on the applied binding mechanism, the user either creates the process model based on concrete service descriptions or based on abstract service templates which are representatives for sets of services (Kowalkiewicz *et*

al, 2008; Fluegge *et al*, 2006). The major drawback in this approach is that, with respect to multitude of available services and service templates, it may be a time-consuming task to manually select reasonable building blocks for the composite service. In addition, the creation of the data flow, i.e. the parameter assignments between the activities, can be complex and might require user to have extensive knowledge about the underlying data type representations.

We employed an *automated* model which overcomes the limitations of the manual composition. Automated process models are more suitable in dynamic domains. This approach generates a service composition plan without human interaction. When a fully-automated process model is combined with runtime service binding, the complete service composition can be performed at runtime. Generally *runtime compositions* are meant to be "throw-away" compositions, which mean they are created for one request, optimized for this request, and later thrown away after the enactment of the request.

2.4.3 Richer Collaboration and Flexible Interaction

Another very important characteristic that differentiates our service composition strategy is in the *execution model*. As discussed earlier, the two execution models commonly used in service composition based on workflow are *orchestration* and *choreography*. However, the majority of workflow researches are based on orchestration, but choreography is receiving attention lately especially in view of the challenges of centralized orchestration model of workflow systems (Baker et al, 2009).

However, Singh and Huhns (2005) present the case for a third execution model based on flexible interaction among software agents. This execution model is *collaboration*. Just like in choreography, collaboration also takes the view of service composition as collaboration among business partners but differs from choreography in that the business partners do not only send messages to one another but also enter into business relationships such as contracts and obligations. They generate flexible message exchanges depending on the evolving circumstances and their local policies, e.g., to handle business exceptions. Collaboration supports a deeper level of interactions and cooperation

among processes. Our service composition execution model follows a collaborative interaction among software agents.

2.4.4. Decentralized Execution Monitoring

The final aspect of our service composition strategy is in monitoring and adaptation to runtime changes. Our system is based on a decentralized execution monitoring. The decentralized execution monitoring process is facilitated by service agents who are in charge of monitoring the grid/web services they are assigned. Decentralized execution monitoring has advantages over conventional centralized system because faults are easily discovered and diagnosed at a faster rate.

2.5. Other Related Works

Although, we have concentrated most of our discussion on current state-of-the-art service composition to workflow techniques because it is the most prevalent approach in practice, however, it is important to point out that in order to eliminate the rigidity of workflow techniques, some other researchers have tried to augment workflow approaches with adaptation mechanisms and we hereby review the relevant ones to our work in this section.

The Rudder project (Li & Parashar, 2006) has some interesting similarities with our work since they also employ multi-agent systems and use tuple-space for coordination. However, Rudder employs agents solely to enact workflow processes, while MINDS is a pure agent-based service composition system. Apart from this, Rudder depends on a central matchmaker for service discovery which make it suffer scalability problem as discussed earlier. Service matchmaking and composition in MINDS is based on decentralized matchmakers in order to achieve scalable and efficient service discovery.

Furthermore, in the work reported by Muller et al (2006), a pure agent-based system was proposed for service composition. While this approach is promising because of its emphasis on emergent behavior through multi-agent system, however, coordination of interacting service agents' is achieved by *coalition formation*. A problem with coordinating agents' activities using coalition

formation (Nwana et al, 1996; Nwana et al, 1998) is computational complexity. The amount of knowledge required of the agent is much because the system assumes that each agent has knowledge of the utility models of other agents. The information shared among communicating agents are much, thereby, increasing bandwidth overhead. MINDS is designed to enable agent operate on limited knowledge and decoupled in time and space so as to optimize communication cost.

Another popular approach to service composition is the use of descriptive (formal) techniques. Many researchers have employed the use of Petri nets, Process Algebra, Finite State Automata to model service composition (Narayanan and Mcllraith, 2002; Tuner and Tan, 2007). The main strength of this approach is that it is good for verification and validation of composition prior to enactment. Errors in the composition are easily detected prior to deployment. The problem with formal techniques, however, is that it can only cater for design-time faults in the composition; there are faults or requirement changes that occur to the system at runtime which requires flexible and adaptive decisions that may not be planned into the system apriori.

Finally, some other research works, have also used Artificial Intelligence (AI) planning approach for service composition in order to achieve automatic composition. For example, Rule-based Planning was employed in SWORD (Ponnekanti and Fox, 2002) where services are represented as rules that are used by the rule-based expert system to devise a concrete plan which can be executed during the composite service request. Although, rule-based planning technique is very useful, SWORD is not based on the Service Oriented Architecture (SOA) standards. Additionally, SWORD possesses weaker expressive capabilities. Another, planning technique, is the Hierarchical Task Network (HTN) planning. Here, the primary objective is to perform certain tasks rather than accomplishing some goals. HTN employs a methodology of planning by task decomposition in which a complex task is broken down into set of primitive tasks that can be assigned to individual processing entities. This technique is employed in the popular SHOP2 planner (Nau et al, 2003). SHOP2 is, however, domain dependent, and assumes a closed world of services implying that to effectively plan with it, a good knowledge of the domain under which the plan is supposed to operate is

required. Furthermore, the Accord Composition Engine (ACE) which is part of the project Automate (Agarwal et al, 2003) is another interesting plan-based composition. The main objective of ACE is to synthesize composition plans dynamically based on available grid services, defined user objectives and constraints. However, ACE does not support services with semantic description. Besides, ACE focus mainly on planning and do not address composition execution and monitoring.

Table 2.1 summarizes the current state-of-the-art service composition strategies with their strength and limitations

2.6 Chapter Summary

This Chapter has presented the background to our research work and justified the need for a new flexible and adaptive service composition strategy. We have presented a motivating scenario based on e-Tourism Collaborative Virtual Organization. Based on the scenario, we discussed current state of the approach to service composition based on workflow and outlined its pitfalls and limitations which include: lack of flexibility, lack of scalability, poor adaptation mechanism, and poor interactivity. We have motivated for our new flexible and adaptive service composition system, called MINDS. The new service composition strategy is not based on predefined sequence of execution or workflow but rather on goals to be achieved by the composition.

We have overviewed of key features of our service composition strategy including:

- matchmaking based on distributed (active) services;
- runtime, automated service composition;
- richer collaborations and flexible interactions, and
- decentralized execution monitoring.

Table 2.1: Summary of Current State-of-the-art approach for Service Composition

Approach	Techniques	Strengths	Limitations
Workflow e.g Rudder(Li & Parashar., 2006) Triana (Triana, 2003), Taverna (2004), myGrid (2004), and JIGSA (2005)	BPML, BPEL, WSCI, GWEL WSFL	*Good for intra-organizational processes and closed settings	*Centralized execution model. *Rigid. *Priori knowledge required
AI Planning e.g ACE (Agarwal et al, 2003) SHOP2 (Nau et al, 2003), SWORD (Ponnekanti and Fox, 2002)	Situation Calculus; HTN Planning; Rule Based Planning	Automatic composition.	*Assume close and static environments. *Depend on domain knowledge.
Formal e.g CRESS (Turner, 2000); Narayanan and Macllraith, 2002; Tuner and Tan, 2007	Petri nets; Process Algebra; Finite State Automata	Verification and Validation.	* Lack adaptation mechanism to runtime changes or fault in the composition.
Multi-agent System e.g Muller et.al (2006)	Coalition Formation	Emergent behavior based on alliance formation.	*Assume all agents know about each other's preferences. *Computational Complexity

We have also discussed some other related work and strategies found in the literature. We could then conclude that, whereas a number of research work have addressed the challenges of service composition, many of the existing solutions are not suitable for open and dynamic setting where additional requirements are imposed on the middleware. We then turn our attention in the following chapters to discussing our solution in details. The next chapter presents MINDS design and architecture and shows how we address the limitations of current state-of-the-art service composition systems discussed in this chapter. This is followed by a proposed life-cycle process for runtime, automated service composition in Chapter 4. Chapter 5 presents our experimental prototype implementation of MINDS architecture for service provisioning applied to the e-Tourism Virtual Collaboration presented in this chapter. Chapter 6 discusses the evaluation of MINDS through empirical analysis and simulation experiments.

Chapter 3

MINDS Conceptual Design and Architecture

The current trend towards Service Oriented Computing (SOC) as an effective software methodology for engineering distributed applications has led to increasing research interest in middleware technologies and platforms. Although, there exists many middleware systems for service composition, which are currently used in a number of projects, most of them are based on procedural techniques where service composition is treated as a workflow and rely on a central orchestration engine for composition enactment. This makes them rigid and inflexible when applied in dynamic settings where new services become available and requirement for composition may change at any time. Therefore, there is the need to complement existing standards and methodologies of Service Oriented Computing (SOC) with some other software engineering paradigms that make services "aware" of their environment. We believe the techniques and methodologies of Agent Oriented Computing offer this complement and helps us capture richer collaboration and flexible interaction among services based on evolving circumstances and local policies. In view of this, the chapter presents MINDS: a middleware infrastructure for distributed service provisioning. MINDS architecture is structured around autonomous "active" services that are able to proactively take runtime composition decisions based on current situation and evolving circumstances. The chapter identifies the set of design criteria required by a service composition platform in an open collaborative services environment; and put MINDS in context within the SOC environment; It further details the conceptual design and architecture of MINDS and overview its key components.

3.1 Introduction

The Internet is currently transiting to a fully-formed computing environment with the added capability to offer services (Woods et al, 2009). In the emerging "Internet of Services", services (web or grid) distributed on the network become the building blocks for composing new applications (Nessi-Grid, 2006; NGG Report, 2006).

Amongst the many challenges to the realization of the vision of "Internet of Services" is the immaturity of techniques for effective service composition (Verity et al, 2008; Woods et al, 2009). As independently-developed services become massive and ubiquitous, the crucial issue that determines the success of the composite application is how well the system is able to put the services together in an efficient manner. After all, most business services do not function in isolation and are best utilized as part of a larger application.

This growing importance of service composition to the vision of a global "Internet of Services (IoS)" is also generating renewed research interest in novel middleware technologies and platforms for SOC (Issarny et al, 2007). As in any other distributed systems, the enactment of service composition demands a direct interaction with middleware, which provides support infrastructure and runtime execution environment for services (Polze and Troger, 2008). The middleware form the base for the service-oriented distributed application (Sun and Blateky, 2004; Tanenbaum and Van Steen, 2007).

While the crucial role of middleware in SOC is evident and many middleware systems for service composition are already deployed in both scientific and business domains, most of them only realize a minimal aspect of SOC (Issarny et al., 2007). Most existing service composition platforms such as EFlow (Casati et al., 2000), DySCo (Piccinelli and Mokrushin, 2001), SELF-SERV(Sheng et al., 2002), Ninja Paths (Chandrasekaran et al., 2000), Fusion (VanderMeer et al., 2003), ICARIS(Tosic et al., 2000), ARGOS(Ambite and Kapoor, 2007), SeGSeC(Fujii and Suda, 2005), Aurora (Marazakis et al., 1997), SPICE ACE(da Silva et al., 2007), SHOP2 (Nau et al, 2003), ACE(Agarwal, 2003), and Rudder (Li and Parashar, 2006), viewed service composition as process-ordering into a workflow that requires a central orchestrating engine to coordinate the control and data flow. But, as the number of services involved in the

composition increases, using a central coordinator will lead to scalability issue and performance degradation (Bhatia, 2005).

Some other research works, such as the one described in Barker 2009a and Barker 2009b make the case for the use of choreography rather than orchestration in order to realize a scalable service composition system. However, while choreography does not require a central orchestration engine but based on simple message passing between interacting services (W3C, 2002a), it suffers critical limitation in its level of interactivity. Services are naturally "passive" and so could not capture runtime changes in its execution environment thereby limiting the system interactions in the face of a failure or runtime requirement changes (Singh and Huhns, 2005).

Addressing the above challenges requires a new middleware architecture that is decentralized and which enables services to be "active" so as to react to runtime changes. In this thesis, we propose MINDS: a **M**iddleware **I**nfrastructure for **D**istributed **S**ervices provisioning, which focuses on addressing the challenges identified above. In MINDS, service composition is viewed as a *collaborative conversation among "active" grid/web services*. Active services are achieved through software agents that programmatically act as wrappers to normal (web or grid) services in the SOC environment. We believe that for services to interact flexibly; adapt to runtime changes; and sense new opportunities that are relevant to their service offerings, their abstractions and design must enable them to do so. Current Web services standard alone do not provide solution to many of the challenges of large scale distributed systems particularly in a dynamically changing environment (Jennings et.al, 2000; Huhns and Singh, 2005). Therefore, there is the need to complement existing standards and methodologies of Service Oriented Computing (SOC) with some other software engineering paradigms that make services "aware" of their environment. We believe the techniques and methodologies of Agent Oriented Computing offer this complement and helps us capture richer collaboration and flexible interaction among services based on evolving circumstances and local policies.

The rest of this chapter is organized as follows: in section 3.2, we identify a set of design criteria required by a service composition platform in an open collaborative services environment and on which MINDS design is structured; thereafter section 3.3

presents the context of MINDS, showing its communication and interaction with other entities in the SOC environment; Section 3.4 discusses the conceptual design and architecture of MINDS; while section 3.5 concludes the chapter.

3.2 Design Criteria

In the context of an open collaborative business services environment such as the one described earlier in the scenario presented in Chapter 2, the following design criteria become desirable in order to achieve a flexible and adaptive service composition platform:

- **autonomous:** Services are provided by different organizations that often have their own local policies and interests. We need a way of preserving that autonomy during service composition and not require services to be subservient to other services or a broker. In essence, we should model services so that they are able to take decisions on participating in a composition on their owners' terms (Singh, 2004; Singh and Huhns, 2005).
- **semantically-described and rich services:** Many service composition systems still rely on syntactic description to discover and select services during composition. Services need to be semantically annotated so as to improve discovery and selection process (Cardoso and Sheth, 2006). Besides, services must also become "rich" and extend into areas like "opportunity sensing" and must evolve from semantic descriptions to semantic awareness (Woods *et al*, 2009). It is when services are designed to be proactive that they can take such initiatives on their own.
- **adaptive and flexible:** Services operate in open environment where there are exceptions and changing (user and runtime) requirements (Singh and Huhns, 2005). Appropriate support mechanism must be incorporated in the design of the middleware platform so that it can adapt to those changes and flexibly react to faults and exceptions.

- **cooperative:** Services should be designed to be cooperative. This is important because services are no island, but work in cooperation with one another during composition (Singh and Huhns, 2005). We need a means of capturing richer interactions among services and how services work together in the awareness of the behavior of other services.
- **scalable:** Current SOC environments are still very rudimentary with just a few participants (Bhatia, 2005). However, the trend in SOC evolution will drive the move towards pervasive service environment where services would become widely available on the Internet and as part of our everyday life. Such trend shows that SOC environments would be more populous and, therefore, requires an infrastructure that is able to scale as the number of services increase (Huhns et al, 2005).
- **deadline-driven**: Most service composition are deadline driven (Barker et al., 2009). In this sense, composition is expected to be finalized within a given time-frame, else it fails to meet the target. The middleware platform must, therefore, ensure that composition is synthesized within the limit of this timeframe.

3.3 The context of MINDS

The above stated criteria are what formed the core principles in the design of MINDS - a middleware infrastructure for distributed services provisioning (Iyilade et al, 2009). However, it is useful to first define the context of MINDS with respect to how it interacts and interfaces with external entities within the service oriented environment. These externally visible properties of MINDS help us to define context and boundaries within which MINDS operate with the Service Oriented Computing environment.

The first kind of interface required by MINDS is that which concerns its interaction with grid/web services. There are two interfaces required in this regard: One, the interface to discover available services in the ecosystem.

MINDS interaction with the services is through the **service registry** where the services available are published. Two, there is also the need to specify how MINDS is going to bind to or invoke services. The syntactic specification of the operations of a service is given in the Web Service Description Language (WSDL) which MINDS depends on. Therefore, in relation to distributed services, the operation of MINDS depends on semantic service information published in the **service registry** for discovery and the WSDL for binding.

The second interface to MINDS is that which it maintains with the human user especially for configuration and administrative logistics. Human users can interact with MINDS through a *portal* interface.

The third kind of interface, though related to the second one is the interface of MINDS with the enterprise application logic or other services using MINDS. MINDS is a utility middleware infrastructure which user only access on-demand to realize a composite application. A web service interface is a good way for application to take advantage of the services of MINDS.

A possible option is to define an Application Programming Interface (API) to MINDS. This would however, constrain the use of MINDS to a particular programming environment and to a particular application usage scenario.

It must be noted, in addition, that MINDS also need appropriate interfaces to interact with other support services available in the service oriented environment. Such services include: *user preference and context repositories*, and *domain ontology*. The domain ontology facilitates sharing of information in understandable terms among collaborating agents in the ecosystem. Figure 3.1 highlights the context of MINDS by showing its interface to other external entities in the service oriented environments.

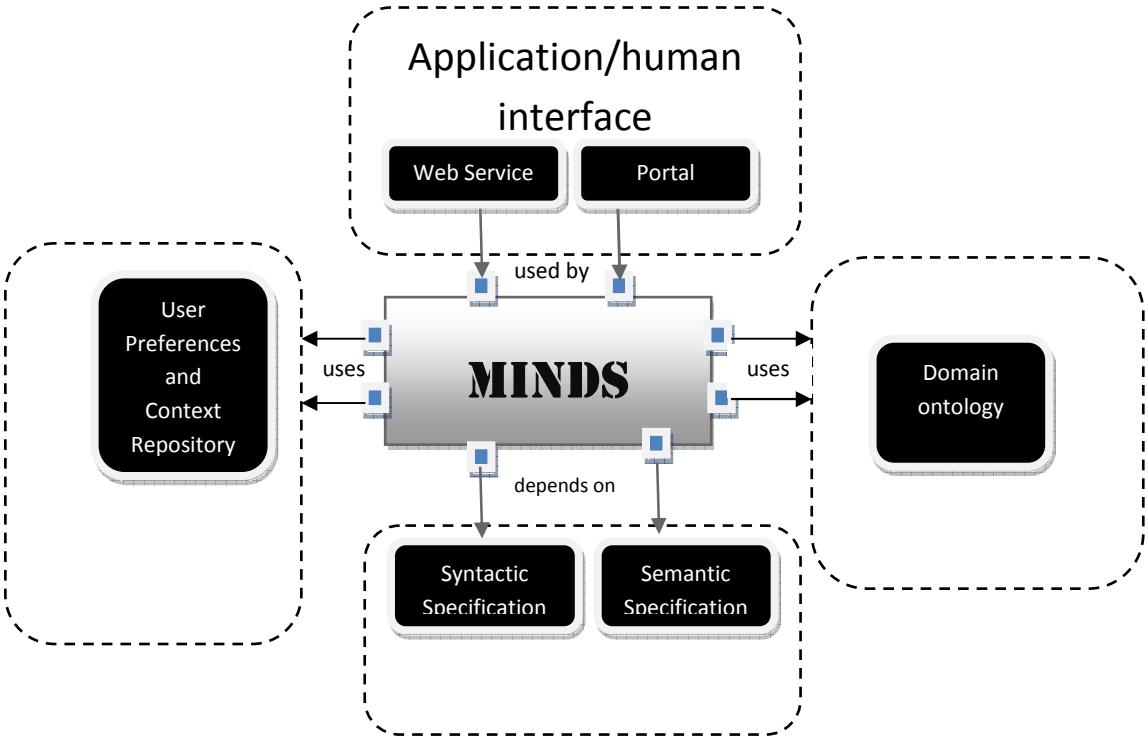

Figure 3.1: Interfaces of MINDS showing its relationship to some external entities within the Service oriented environment

3.4 System Design and Architecture

MINDS architecture, as shown in Figure 3.2, consists of a set of basic components which realize a flexible and adaptive middleware infrastructure for distributed services provisioning. The architecture is based on the conceptual view of collaborative virtual organizations (or virtual enterprises) as dynamically formed teams of agents with a collective goal (Foster et al, 2004). Our approach is to leverage existing standards and trends in SOC (Grid and Web Services) as much as possible and augment with methodologies and advances from Software Agent in critical areas. Agents operate on local (constrained) knowledge and communicate (and coordinate their activities) through an associative shared memory (tuple space). In MINDS, three kinds of

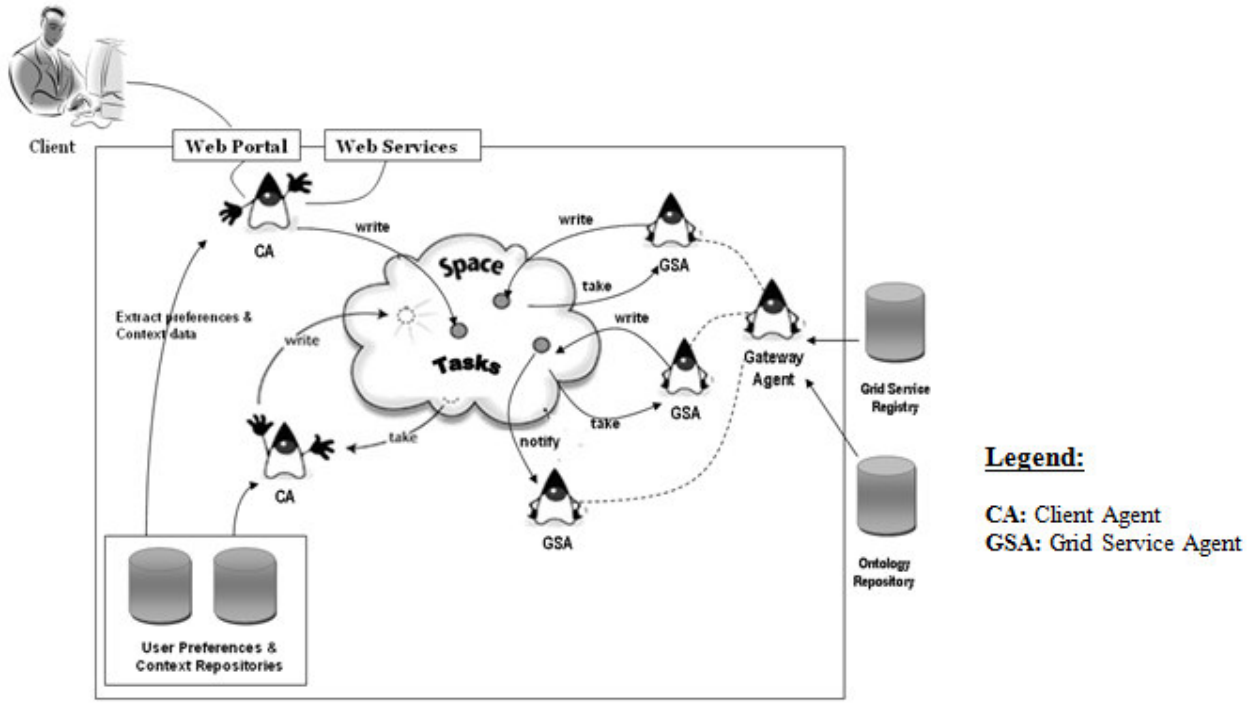

Figure 3.2: Proposed MINDS Architecture

agents are identified, namely: *the Client Agent (CA), Grid Service Agent (GSA)* and *the Gateway Agent (GA)*. The GSAs are built as "wrappers" on top of Semantic Grid services in order to make them active. MINDS also include other support components such as Grid Service Registry (GSR), Ontology Registry (OR), and User Preference and Context Repository (UPCR).

Before going into a detailed description of MINDS components and their role in the architecture, we first discuss some crucial decisions that MINDS design were based upon.

3.4.1 MINDS Design Decisions

In view of our aim towards a flexible and adaptive middleware infrastructure and the design criteria stated earlier, we have made some design decisions to come up with MINDS as discussed below:

(a) ***Grid services as service implementation technology:***

SOC have been recognized as the most suitable technique for engineering distributed applications than previous paradigms such as Object Oriented Programming (OOP) and Component-based Software Engineering (CBSE) (Bhatia, 2005; Singh, 2004). However, a very important choice is the implementation technology for the SOC idea.

Two prominent technologies that are often used are: *Web Services* and *Grid Services* (Foster and Kesselman,1998; Foster et al, 2001; Foster, 2005). The two however, have technically converged but only differ currently in the application focus (Koehler and Alonso, 2007). The new grid computing architecture based on OGSA[7] leverages existing web services standards but only extends it in state management (Tuecke et al, 2002; GGF, 2002; Sotomayor and Childers, 2006). The concept of state, in our view, is a very important attraction of the grid services concept especially because of its usefulness in having a fault-tolerance system. The implication of this is that if there is failure during a service execution, we can capture the state of the job and assign it to another provider who can use the state information to continue from the point of failure. But, even though we have chosen the grid services as our primary implementation technology for services, we utilize a number of web services standards and frameworks in our implementation.

(b) ***Modeling collaborative virtual organizations as dynamically formed team of agents:***

Another crucial decision we made in MINDS design is to employ models and techniques of Multi- Agent Systems in analyzing and studying the interaction among (business) entities in the virtual

[7] OGSA – Open Grid Services Architecture

organization. Even though, service technologies provide standard ways to model communication among distributed services, they suffer limitations in many areas (Singh and Huhns, 2005). For example, services are naturally "passive" unless they are invoked and thus cannot proactively sense opportunities. Apart from this, we need a way in which to preserve the autonomy of the service providers, which means they must be able to decide on their own terms which composition to participate in. So, we have to look at other software engineering methodologies that could complement current SOC standards (IBM, n.d) in these crucial areas. Research in the area of Agent Oriented Software Engineering have models and methodologies that in our own opinion provides this complementary role.

Software agents naturally possess the following properties which make them suitable for realizing our goal (Woolridge and Jennings, 1995; Bradshaw, 1997; Jennings and Woolridge, 1998; Weiss, 1999; Stone and Veloso, 2000; Wang, 2007):

- *Autonomy:* Agents operate without the direct intervention of human or others, and have some kind of control over their own actions and internal state (Castelfranchi, et al, 1992). Autonomy of agents provides the best way of modeling the behavior of service providers (who are, in reality, autonomous).
- *Social ability:* Agents interact with each other (and possibly humans) (Genesereth & Ketchpel, 1994). In view of this, agent can enter into collaboration with other agents and can operate asynchronously and in parallel on any task to be addressed. This can result in an increased overall speed and system efficiency.
- *Reactivity:* Agents perceive their environment (e.g the physical world, or the Internet) and respond in a timely fashion to changes that occur in

it. This perception allows agent to flexibly adjust to exceptions and faults that might occur in their operating environments.

- *Pro-activeness:* Agents do not simply act in response to their environment; they are able to exhibit goal-directed behavior. This quality of agents makes it possible for agents to have semantic-awareness and respond to opportunities.
- *Adaptability:* Agent teams are dynamically formed. Such teams, however, can be easily reconfigured and disbanded in view of changing requirements (Cao, 2001).
- *Robustness and Reliability*— The failure of one or several agents does not necessarily make the overall system useless, because other agents already available in the system may take over their part.
- *Scalability:* The decentralized nature of multi-agent systems make them overcomes the performance overhead often experienced in SOC due to scalability. The system can scale to an increased problem size by adding new agents, and this does not necessarily affect the operation of the other agents.

Finally, in order to take advantage of the strengths of both the SOC and AOSE, we have decided to "wrap" semantic (grid) services with agents so that the internal communication and interaction between the business entities takes place using Agent Communication Languages (ACL) (DAML, 2001).

(c) Associative Shared Memory for Coordination:

A very important decision in any distributed system is how to flexibly coordinate the activities of the distributed components. Coordination has been studied by researchers in several fields including distributed artificial intelligence, social sciences, political science, social psychology, anthropology, law and sociology (Nwana et al, 1996). In order to facilitate

effective communications and coordination among agents, we employed associative shared memory (tuple space) coordination model which was made popular by the Linda Coordination Language (Gelenter, 1985). The reasons for our choice of Associative Shared Memory are (Freeman et al, 1999):

- *decoupled communication:* its model allows communications among agents to be decoupled in both space and time. Hence, an agent only have local and constrained knowledge about the other entities in the ecosystem it is communicating with;
- *search by pattern matching:* Entities can be located by using pattern matching (not just simple keywords). This provides a simple means of finding the objects that is of interest, according to their content, without having to know what the object is called, who has it, who created it, or where it is stored.
- *simplicity:* The tuple space allows published entities to be accessed through simple operations such as *write*, *take*, *read*, and *notify.*

3.4.2 Overview of MINDS Components

Having outlined the design decisions that we have made in MINDS, we now provide a high-level overview of its components. As stated earlier, MINDS consists of a set of components that together realize a flexible and adaptive middleware infrastructure for distributed service provisioning. The architecture is realized through the following set of basic components:

- *Client Agent (CA):* The Client Agent collects user requests; decomposing them (if necessary) and publishing the tasks on the tuple space. It further synthesizes composition plans and present results of composition back to the user.
- *Grid Service Agents (GSAs):* Grid Service Agents are realized as agent "wrappers" on Semantic Grid Services thereby making the services "active" and "alive" to be able to proactively take decisions. The

responsibilities of the GSA include: proactively searching for user task to participate in; placing interest in service offerings on the discovered task; task execution; and monitoring task execution.

- *Gateway Agent (GA):* The main role of the Gateway Agent is to dynamically generate the GSA agent class including its associated proxy element upon new service registrations in the service registry by service providers. The GSA accomplishes these by constantly polling of the service registry to identify new services that comes in.
- *Tuple Space:* This part of MINDS handles the communication and cooperation between the agents. The tuple space (TSpace) provides agents with a flexible coordination model that decouples them both in space and time. The TSpace is used by the CA to publish tasks and its execution plans. Also, GSAs use the TSpace to leave service advertisement on tasks and also to publish results of its execution.
- *Grid Service Registry (GSR):* This is an external directory that keeps information on grid service registrations. The GSR is a registry such as the GridMDS or the UDDI that enables providers to register their service offerings.
- *Ontology Repository (OR):* MINDS also uses a domain ontology repository that keeps ontological description of published services.
- *User Preferences and Context Repositories (UPCR):* In MINDS, users request are often submitted with quality of service specification. User preferences and context are preserved in the UPCR so that they are used for customization and personalization of responses.

We will then provide further detail descriptions of the core components of the MINDS architecture – the Client and Grid Service Agents.

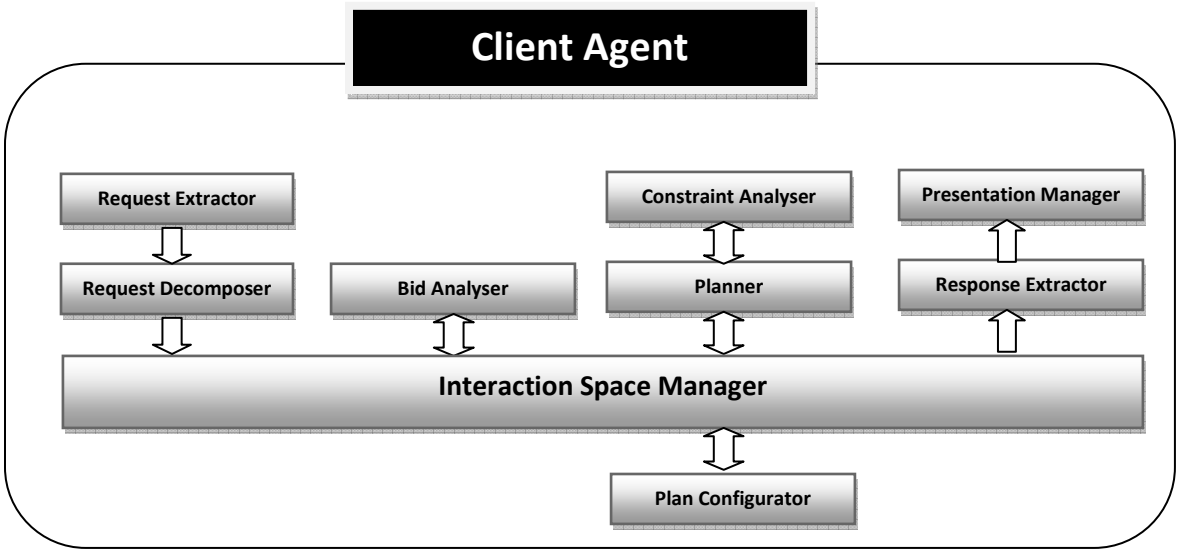

Figure 3.3: Client Agent Components

3.4.3 Details of MINDS Components

3.4.3.1 Client Agent (CA)

The component of the client agent is as shown in Figure 3.3. The CA roles in MINDS include:

- interfacing with the user for request;
- bid analysis and task award;
- planning for task execution in the tuple space;
- generation of response to the user.

(a) User Request Extraction and Analysis

The CA is responsible for interacting with the user application logic for user request input. During the request operations, the CA extract client request for task to be performed through its ***Request Extractor (RE)*** component. The Request Extractor uses appropriate protocols such as SOAP[8] to achieve this. Thereafter, it analyzes the request through its ***Request Analyzer (RA)***. The

[8] SOAP – Simple Object Access Protocol

analysis is to determine if the task is simple or complex. If the task is complex then it needs to be decomposed into simpler tasks before it is then published in the tuple space.

(b) Bid Analysis

The CA is also responsible for managing the bidding process that occurs during the selection of the service provider. This is achieved through the ***Bid Analyzer (BA).*** The bid analyzer uses the Contract Net Protocol (CNP) to advertise and select appropriate service provider for a particular task. In the bidding process the steps that a CA is expected to take are:

- publish a task that needs to be performed in the tuple space;
- receive and evaluate bids from potential service providers;
- award the task to a suitable contractor;
- receive and synthesize results

Similarly, from the perspective of the service providers (represented by the Grid Service Agents), the steps in the bidding process follow:

- listen for and receive task announcements;
- evaluate capability to offer service;
- respond (decline, bid)

(a) Planning

Planning is the reasoning side of acting. In order to execute the tasks, the CA generates a partial plan for each of the Grid Service Agents (GSAs) so that they can start the task that is assigned to them. As against human planners who may not have a global understanding of how the whole services fit together, automated planners (Agarwal, 2003, Zeng et al, 2003; Pistore et al., 2004; Sirin et al, 2004; Berardi et al, 2005) plan according to an algorithmic planning strategy such as for example forward- or backward chaining of services. The system thus suggests a partial plan for composition. We adapted

the graph-based automated planning algorithm presented in Agarwal (2003) for our service composition. The planning algorithm is presented in Figure 3.4.

1	Given Service Graph **G(S,T),** where, **S** is the set of available services and **I** set of possible interactions (b) Service set **S** = $\{s_i\}$ and each s_i is associated with an ordered set of keywords, $\{K(s_i)\}$ (c) Interaction set **T** = $\{\mathbf{t_{i,j}}\}$ such that $s_i, s_j \in S$. Each interaction $\{\mathbf{t_{i,j}}\}$ has a cost value *Cost* ($\mathbf{t_{i,j}}$) associated with it.
2	*Create Global Plan:* Within the service Graph, G(S,T), the available services are vertices and interactions are edges. The edges are created at runtime using relational join operation: $t_{i,j} \in s_i \bowtie s_j$ $(i.e\ find\ if\ output\ of\ s_i\ can\ be\ chained\ to\ the\ input\ of\ s_j$
3	The CA specifies composition as a set of s_{initial} and s_{final} and a set of constraints $\{c_k\}$, an ordered set of keywords, $\{K_{conposition}\}$
4	*Create a Local Plan call composition graph* ***G'(S',T')*** *:* $\forall\, i, s_i \in S' \iff K(s_i) \subseteq \{K_{composition}\}$ $\forall\, i,j,t_{i,j} \in T' \iff s_i \in S'\ and\ Valid(t_{i,j},\{c_k\}) = True$
5	Dynamic Service Composition = finding a path from s_{initial} and s_{final} in G'(S',T')

Figure 3.4. Composition Plan Generation algorithm (*Adapted from Agarwal, 2003)*

The CA generates the task execution plan based on what is available at runtime. That is, the plan is generated based on available services and user quality of service constraints (such as *price* and *deadline*). In MINDS, the service composition is addressed as a *graph planning* problem. Available pool

of services is represented as a graph in which nodes are the services in the pool and the links define the interactions and the feasibility of the composition.

During planning, the CA first collates the constraints imposed on the composition plan such as (price and deadline) based on request analysis phase. This is achieved using the *constraint analyzer* module. Then a plan is generated using the *planner* module. The planner uses the graph planning algorithm to general task execution plan. Finally, the *plan configurator* then chains the various nodes in the service graph together using relational join. Figure 3.4 shows the steps involved in the composition plan generation and configuration

(a) Generation of Response to the User

During response operation the CA presents initial result of composition to the end-user for verification and permission to initiate execution. All through the instances of generating responses to the client, the CA relies on appropriate context and preference information about the user to provide response in appropriate format. For example, where client is a browser, the response is formatted in HTML[9] and if the user is using a mobile device, the response can be generated as a J2ME[10] file.

The first step during the response is to extract the response into a single file (basically as an XML file) using the ***Response Extractor (RE)*** component and then use a ***Presentation Manager (PM)*** to translate the response to the appropriate context and format of the end-user.

[9] HTML – Hypertext Mark Up Language
[10] Java 2 Micro Edition

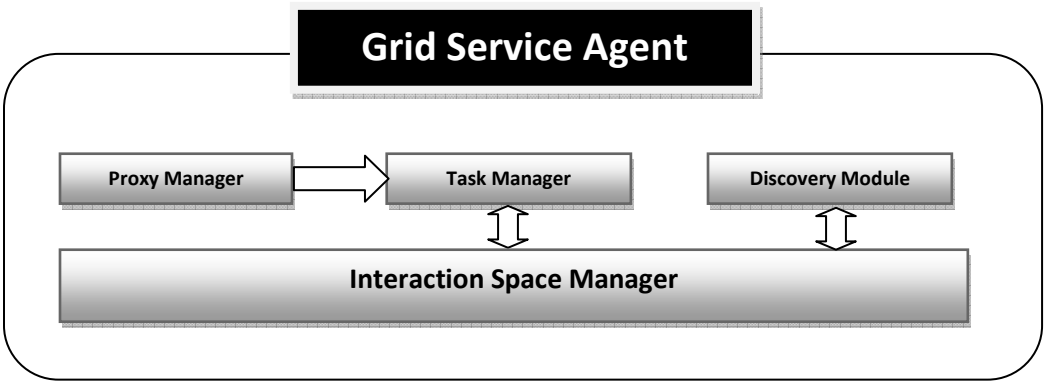

Figure 3.5: Components of the Grid Service Agent

3.4.3.2 Grid Service Agent (GSA)

The Grid Service Agent is the agent that is created to represent each provider of services registered in the service registry. The functional component of GSA is as shown in Figure 3.5.

The *Discovery Module* of the Grid Service Agent (GSA) is responsible for proactively searching the tuple space for user task to participate in based on their local semantic service descriptions and policies. When it finds a matching service that is of interest, it then places a bid in offering the service required of the discovered task on the tuple space. In this context, we have employed the GSAs as distributed matchmakers who could indicate interest in any task of interest to them. Unlike conventional approach where the services are passive and could not participate in any task unless invoked, this approach is desirable on three counts:

- **autonomy of provider is preserved:** That is, because services are not made to be subservient to the broker or any other service invoking it. Service autonomy is preserved since it could decide on which task to participate in based on its semantic description and local policies. For example, if a task would violate a particular local policy of the organization that provides the service, the GSA for that service could simply withdraw interest in such task.

- **increasing richness**: Although the services considered in the context of MINDS are semantically described, using GSA as semantic and context reasoners ensure semantic awareness and richness of the entire system. This flexibility, which we refer to as "opportunity sensing", ensures that services that are not even known at design time can add value to the composition and be dynamically incorporated into the composition process.
- **time for discovery would be minimized:** As the number of services that are available in the service registry increases so also is the time it takes to discover a matching service to a particular task to be performed. We believe that since each GSA proactively listens to the global tuple space to discover tasks that matches its semantic service description and does not violate its local policies, this would, obviously, reduce the service discovery time.

The *task manager* component of the GSA is responsible for receiving task awarded to the GSA and accepts all input data and publishing output data back into the tuple space. When a task is assigned to the GSA, it invokes the grid service in order to execute it by using its *proxy manager*. GSA are however, generated through a gateway agent which periodically monitors published task in the registry.

3.5 Chapter Summary and Comments

In this chapter, we have discussed the crucial role of a middleware infrastructure in SOC and its particular important in service composition. We have identified the need for a good middleware design and architecture in order to facilitate effective composition strategy. We have outlined the limitations of existing service composition strategies especially when applied in dynamic settings and emphasized the need for a more flexible and adaptive service infrastructure. Subsequently, we have presented the design and architecture of MINDS: a middleware infrastructure for distributed services provisioning developed in this

thesis. We have, first, started by outlining the set of design criteria on which MINDS is structured. Thereafter, we have stated the context of MINDS and key design considerations that was made in formulating MINDS. We have then presented description of MINDS architecture and overviewed its main components.

It is, however, important to further examine how MINDS really achieve its service composition process. While, this chapter has focused on elaborating on the components that made up MINDS architecture, it will be necessary to discuss the process (or steps) involved in MINDS service provisioning. To achieve this, we turn our attention in the next Chapter (Chapter 4) on formulating a service provisioning life-cycle based on MINDS architecture presented in this chapter. This enables us discuss details of how service composition is achieved in MINDS and thereby highlighting some further contributions made to current state-of-the-art service composition.

In addition to this, it is also crucial to evaluate the MINDS architecture as discussed in this chapter to demonstrate its applicability in a real-life scenario and verify its performance benefits / tradeoffs when compared with existing strategies for achieving the same goal. These are the focus of our discussions in Chapters 5 and 6 respectively.

Chapter 4

MINDS Service Provisioning Life-cycle

Software application development through service composition differs from conventional software development methodology. The life-cycle process involved in provisioning services as part of a distributed application involves service discovery, selection, composition and binding. Some or all of these steps could be performed at *design time* or at *runtime*. In addition, some of the steps could be carried out manually by human, or they may be semi- or fully automated by the system. Currently, most service composition system requires the user to manually select the services that will participate in a composition with the aid of an advanced composition tool. This, however, could be time-consuming, laborious and error-prone when the number of services to be selected from is massive. Hence, the desirability of an automated, runtime service composition, where composition is tailored and optimized for a specific user request. This chapter proposes a service provisioning life-cycle process that seeks to achieve runtime, automated service composition. The life-cycle process highlights service composition strategy based on the MINDS architecture discussed earlier in Chapter 3. Contrary to some existing techniques, MINDS service provisioning process is not based on a predefined order of service execution but on user request and preconditions. The process starts with *user request* and terminates when *composition results* are generated to the end-user. In-between is a four-step process for *task composition and representation*; *dynamic service matchmaking*; *composition synthesis*; *execution and monitoring.* The system also includes mechanism to react dynamically to changes in user and system requirements.

4.1 Introduction

Developing application using Service Oriented Computing methodology differs from conventional software engineering methodology where application modules and components are developed from the scratch by the software developer or bought off-the-shelf from vendors and installed locally (Mendosa, 2007; Lee *et al*, 2003). A major attraction of the SOC methodology is that it enables effective reuse of existing computational elements within and across enterprises (NGG Report, 2006; Nessi-Grid, 2006). Hence, to develop distributed applications, a service-oriented methodology replaces traditional code generation with a combination of service discovery, selection, composition and binding (Huhns and Singh, 2005).

Generally, in most service composition, the following steps are taken[11] (Fluegge *et al*, 2006):

(i) concrete services to be bound to the process activities are discovered;

(ii) during invocation of a composite service, a coordinating entity (e.g. a process execution engine) may manage the control flow and the data flow according to the specified process model.

(iii) a process model specifying control and data flow among the activities has to be created, and

(iv) the composite service must be made available to potential clients;

However, some or all of these steps might occur at design or run time; they may also require human involvement or not. Hence, the activities and steps involved in engineering a service-based application are often distinguished by the *point in time* of their composition and the *degree of automation* of the process.

[11] The steps are not necessarily in the particular order they are listed

Conventionally, many service composition platforms in use today are based on workflow techniques and adopt a service provisioning process that is based on *manual, design time composition* (Kowalkiewicz, 2008; Fluegge *et al*, 2006; Alamri et al, 2006). In manual, design time composition, a human process designer is involved in creating the composition at design time. The human user is required to define the data and control flow of a composite service manually, either directly or by means of designer tools. Subsequently, the process description is then deployed in a process execution engine (Juric, 2006). However, in open and dynamic settings where the number of services involved in the composition can be massive and composition (business or user) requirement changes are the norm, manually composing services at design time can be time-consuming and error prone (Kowalkiewicz, 2008). The implication of manual composition is that, when new services are available, existing service compositions might be inadequate or even incorrect. Hence, it becomes practically difficult to manually and regularly transcribe new service offering into the composition model.

Shifting service compositions from design time to run-time and from manual to automated approaches promise to solve this problem (Kowalkiewicz, 2008). In this case, no human modeler is involved in creating the composition; however, composition is synthesized at runtime and optimized to meet a specific user request on-demand.

In this chapter, we propose a life-cycle process for automated, runtime composition based on MINDS architecture presented earlier in chapter 3. Unlike in conventional workflow processes, which requires a manually defined control and data flow by the human user, MINDS service provisioning process is based on just preconditions and goal of the composition as specified by the user request. As shown in Figure 4.1, automated, runtime service provisioning process based on MINDS comprises of a series of steps that begins with a client request and terminates at composition results generation by the platform. In-between is a four-step process for *task composition and representation*; *dynamic service matchmaking*; *composition synthesis*; *execution and monitoring.* Besides, the system also reacts dynamically to

changes in user and system requirements. The rest of the chapter discusses the detail of the processes involved in each of these steps.

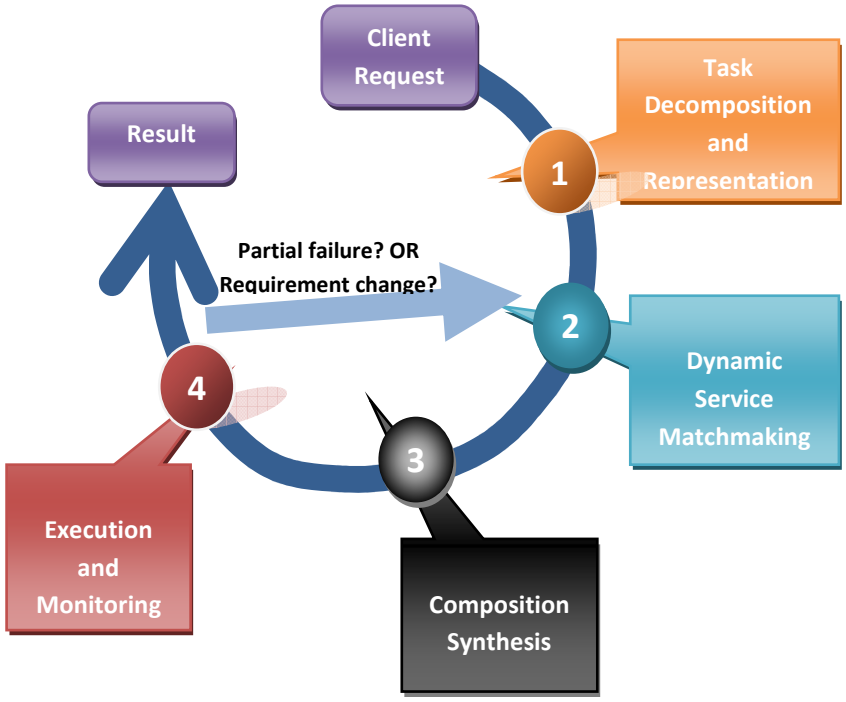

Figure 4.1: MINDS Service Provisioning Life-Cycle

4.2 Client Request

The entry or initial point of this service provisioning life-cycle is a service request by the client to the middleware platform. Unlike in traditional approach, where such request includes a description of what shall be achieved and which concrete service has to be executed during the composition (since composition is planned in advance), a client request in MINDS is rather a specification of the task to be performed and the constraints (such as, the quality of service (QoS) preferences of the end-user, e.g price) associated with it.

A *task,* in this instance, is a high level goal or objective that an end user wants to achieve. This is often specified as high level or abstract goal. The task can be *simple* or *complex*. A simple task can be performed by a single service provider, while a complex task need to be further decomposed into simpler ones before they are

assigned to appropriate processing entities or service providers[12]. For simplicity, we assume each simple task can be mapped to a single atomic service.

The distinction between a task and a service can, therefore, be summarized thus: *a task describes what the user wants to achieve including the constraints that are associated with it, while a service deals with how the goal is realized under the given constraints.*

If we refer to the Planit use case scenario discussed in Chapter 2, *Joe* can make a request to book his business trip directly through the *Planit web interface* by filling a request form on *Planit* web site or through a mobile interface connected to the Planit Order Receiver. In this case, *Joe* is able to fill out information on his preferences alongside with the request in the form. The *Planit portal application* then translates the form entries into a request (task) for processing by the MINDS middleware.

A client request (task), formulated in natural language, might be[13]:

> *I need a **flight** that is **less than R1500** from **Durban** to **Johannesburg** in the **next one week**, is such a flight **available?***

Or in a more complex scenario, a client request might look like this:

> *Given **Joe** with **VISA credit card number 3216187978360999**, and **expiry date** of **1st of December, 2012,** who want to go on a business trip to from **Durban** to **Sandton, Johannesburg** in the **next One week**, **book a flight** that is **less that R1500**, **reserve an executive car** and also **reserve a hotel** that is **near** to the location of his business meeting*

[12] Note that complex tasks are not decomposed by the client but by the MINDS infrastructure, this is performed in the first step of the life-cycle that is described subsequently.

[13] Important keywords in this request are in bold print and represents the domain vocabulary or constraints on request

4.3 Task Decomposition and Representation

The client request is sent to MINDS through the Client Agent (CA). The CA will then accept it together with the user preferences.

If the task is a complex task, the first action of the CA is to decompose it into simpler tasks for representation in the TupleSpace. The CA relies on domain ontology in order to carry out the decomposition. The main purpose of the domain ontology is to capture the concepts, relations, instances, and axioms of the chosen domain. However, since the conceptual design of MINDS is to make it generic, the domain ontology is not included as a main component in MINDS but as an external component that it relies on for task decomposition by the CA.

The CA parses the task and partitions it into a set of sub-requests represented as a pair of task, and associated constraints.

In the Planit use-case, the first request presented above partitioned into tasks might look like this:

<**find flight** | **price** < 1500 | **within week** = 1 | **location from** = DRB **to** = JHB>

For the second example of user request presented in Section 4.2, the request can be partitioned into tasks as follows:

<**book flight** | **price** < 1500 | **within week** = 1 | **location from** = DRB **to** = JHB>
<**reserve car** | **class** = executive>
<**book hotel** | **area** = Sandton | **city** = JHB >
<**charge user** | **name** = Joe | **card type** = VISA | **card holder** = Joe | **card number** = 3216187978360999 | **expiry** = 01-12-2012>

These sub-tasks are then represented as an object of type **Task** which consists of attributes such as **task_id**, **task_desc**, **task_status**, and **plan_id**.

Where,

- **task_id** is a unique identifier that the CA uses to identify each task that it published in the tuple space;
- **task_desc** is a short description of what the task is about;
- **task_status** is an indication of the status of the task. At this stage the status of the task is "**available"**;
- **plan_id** is a unique identifier for the partial execution plan for this task.

After successful decomposition and representation, the CA will then dynamically create its own local interaction space and then publishes the sub-tasks along with their associated constraints into the tuple space. The CA, however, maintains the identity of its created interaction space.

4.4 Dynamic Service Matchmaking

Unlike conventional approach where services are passive and have to wait until they are invoked by a service broker or consumer, service matchmaking in MINDS is achieved through a set of "active" grid/web services implemented as distributed matchmakers. The matchmakers pro-actively search the tuple space for tasks that they can contribute to. The distributed matchmakers are implemented as grid services wrapped inside agents (called GSAs).

We assume that a GSA is given a single task, therefore, a GSA cannot perform multiple tasks simultaneously. During the startup of MINDS platform, the Gateway Agent creates all GSAs based on available services and deploys them to the MINDS platform. Each created GSA consists of the running agent code, a semantic description and the bindings of its associated services. The GA creates the GSA by listening to different grid service registries and upon

registration of a new service in the global registries, it creates a new local GSA to represent the service.

If we consider the Planit use case again, as soon as the CA publishes *Joe*'s request as a simple primitive task in the tuple space, the GSAs will start matching their semantic service descriptions with the requirements of the published tasks. For example, in the tourism business ecosystem considered in this use-case, there are HotelGSAs representing different hotel services; FlightGSAs representing different flight services and CarHireGSAs representing different car rental services. Each of these services will search

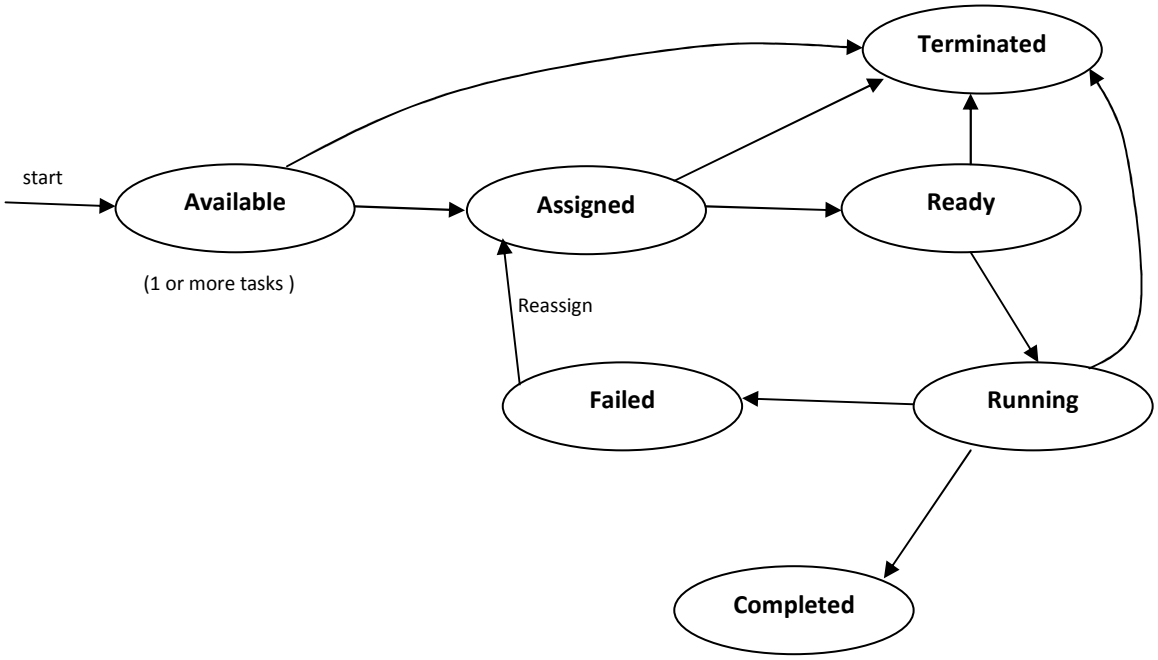

Figure 4.2: A state-transition diagram for published tasks in the tuple space

the tuple space in order to find a task that best suits its semantic service description. More specifically, a FlightGSA will search the tuple space to locate the <*book flight*> task that fit its semantic service description. If a GSA finds a task that matches its criteria, it will do the following:

(i) bid for the task;

(ii) if assigned the task by CA, change the task status in the space from "available" to "assigned"[14];

(iii) query the **task_id**, **plan_id** and **result_id** of the task;

(iv) leave its service description on the task in the space; and

(v) subscribe for a notification and wait for the task execution plan to be available in order to start executing the task.

It must be noted that if the status of the task in the tuple space is **"assigned"**, other GSAs will assume that the task is occupied and not attend to it. Other transitions during life-cycle are covered in the rest of the sections. Figure 4.2 shows a state-transition diagram for status of tasks through-out the service provisioning life-cycle.

4.5 Dynamic Composition Synthesis

The next phase following the matchmaking process is dynamic composition synthesis. The CA will synthesize execution plans for each GSA based on the published semantic service descriptions on tasks. Execution plans are synthesized by creating a graph which maps the inputs and outputs of all services which have left their semantic descriptions on published tasks. Thereafter, the graph is partitioned into partial plans, which are then published into a local tuple space for consumption by the respective GSAs involved in the composition[15].

In this case, all the GSAs will coordinate the execution flow of composite service using their partial plans. Besides, unlike conventional approaches that statically bind to services at design time, the MINDS composition model achieves flexibility by synthesizing composition plans based on available services at runtime. After plans for tasks are published on

[14] We already describe the CNP protocol used for bidding and task assignment in Chapter 3
[15] Graph Algorithm used for generating the global and local (partial) execution plan was presented in Chapter 3

the tuple space, the status of each task then changes from **"assigned"** to **"ready"**.

4.6. Task Execution and Monitoring

The final stage of the service provisioning life-cycle process is task execution and monitoring. On getting the partial execution plan, each GSA will initiate and monitor execution of the service it represents. Therefore, the execution of the service provided by the GSA starts just after the GSA has retrieved the execution plan corresponding to its assigned task from the tuple space. All GSAs will receive their execution plans by periodically checking the tuple space for plan availability using the plan_id. Hence, if the GSA retrieves a valid plan that corresponds to its assigned task, it will first check the plan for dependencies that needs to be satisfied before task execution.

This includes the execution policies that define how the task should be adapted during execution. If there are no dependencies defined in its plan description, the GSA will check whether there are any notifications or instructions it is supposed to receive before initiating execution. If there are no messages or instructions, the GSA will change the status of the task on the tuple space from "ready" to "running" and immediately start task execution by invoking the corresponding service. However, if there are task dependencies, the GSA will first check the execution policies to determine whether execution can commence without waiting for task dependencies. If it is possible, the GSA will commence on task execution, otherwise it will wait until the relevant dependencies becomes available. Finally, the GSA will publish the results of a successful service execution into a tuple space as a result entry which is identified by a result_id associated with the completed task.

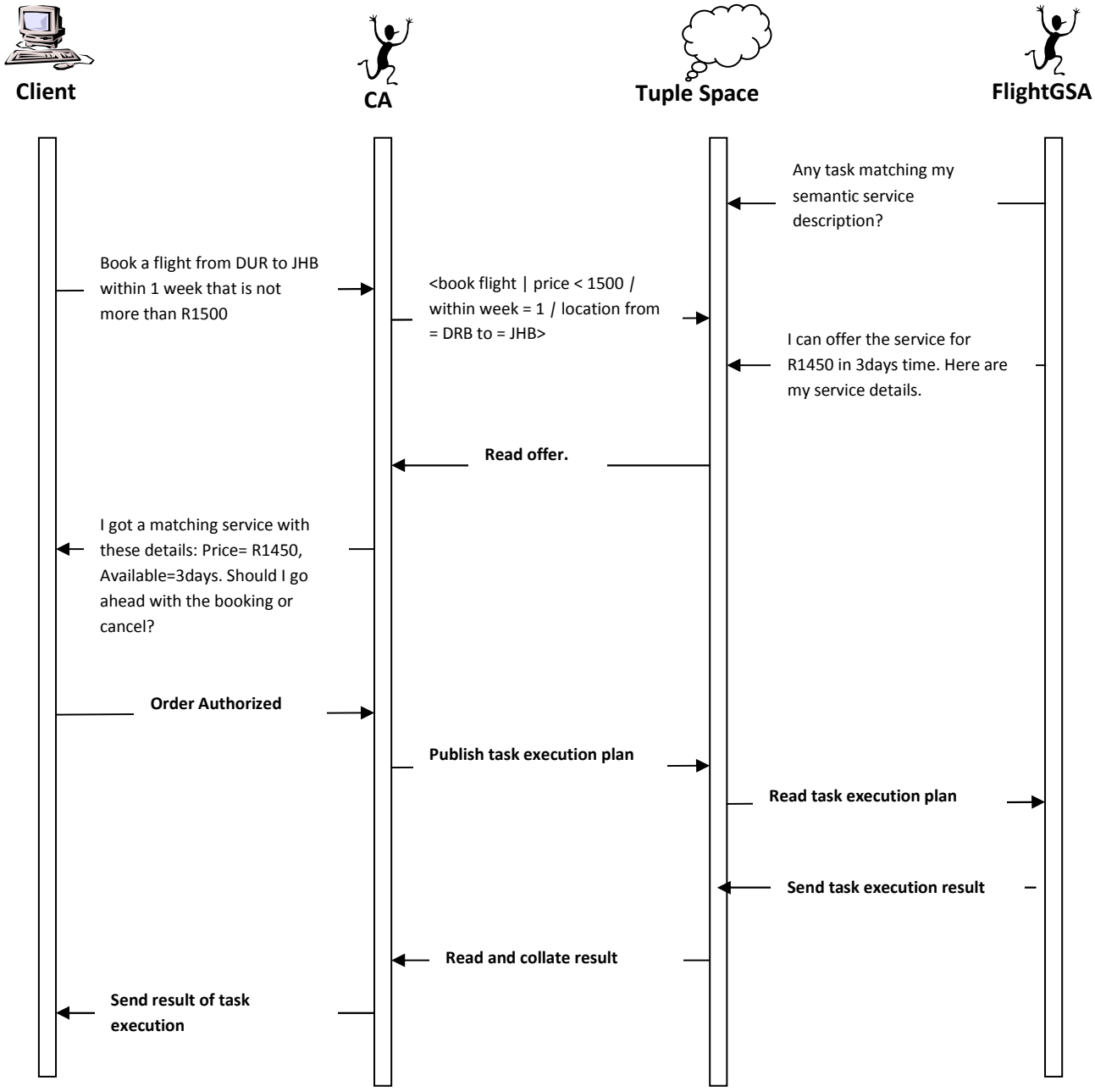

Figure 4.3: Interactions in virtual enterprise collaboration based on MINDS life cycle

We illustrate the kind of interactions that often occur in virtual enterprise collaborations and how such interactions are enabled by MINDS in Figure 4.3.

4.7 Dealing with failure and requirement changes

During the execution stage, a service might fail while executing its assigned task due to exceptions in its internal logic and become temporarily unavailable. Such a partial failure is the norm rather than exceptions in a distributed system environment. In such a case, the GSA representing the service will autonomously adapt by modifying the status of its assigned task from "running" to "failed." It will also query the execution state of the service and publish it into the tuple space. The execution state can then be made available to any other waiting GSA in the same service category so as to continue from the point of failure.

In addition, the system also adapts composition to changes in user requirements at runtime. As an example, if *Joe* requires the *travel service* to include *weather service*, this could be dynamically incorporated to the running composite service since the service composition does not depend on a predefined workflow. In another case, if *Joe* decided to cancel *car booking* service because he has already spoken to a friend to pick him up at the airport, the *CA* may send instruction to terminate the car booking while allowing the rest of the services to continue execution unimpeded. In fact, the continuation of the task can be terminated at any time by the CA during the life-cycle when the task is available, assigned, ready or running. Although, depending on the stage in the execution where the task is terminated, there may be some penalty to the requester/client for cancellation according to normal terms and conditions of the provider. A terminated task has its status changed to "terminated."

4.8 Chapter Summary

In this chapter we have presented our own service provisioning life-cycle process for runtime, automated service composition based on the MINDS architecture discussed in chapter 3. The life-cycle process defined the four-step strategy for service provisioning including *task decomposition and representation*; *dynamic service matchmaking; dynamic composition synthesis; execution and monitoring*. We have discussed each of these steps in detail and illustrated with examples on how they deal with service provisioning for the Planit use-case that was presented in chapter 2.

In the next chapter, we will turn to discussing an experimental implementation of the life-cycle process to demonstrate its practical applicability in a real-life setting and as a proof of concept.

Chapter 5

MINDS Prototype Implementation

In order to demonstrate the utility and applicability of MINDS middleware in a real-life scenario for addressing automated, runtime service composition, and as a proof-of-concept, this chapter presents an experimental prototype implementation of MINDS architecture for the itinerary planner use-case scenario presented earlier in Chapter 2. It is, however, important to note that, although, the e-Tourism itinerary use-case scenario provided a realistic example of service composition problem that MINDS seek to address, the concepts discussed in this thesis applies into a lot of service composition problems found in many other domains. The chapter starts with an overview of the prototype implementation setup, describing the various platforms and frameworks used in the experiment and the container services on which MINDS was built. Thereafter, we carry out a walkthrough of the Planit experimental implementation using MINDS. Section 5.3 gives a summary of the chapter.

5.1 Implementation Overview

We have carried out a prototype implementation of MINDS architecture to demonstrate its applicability and as a proof of concept. The aim of the implementation was to illustrate runtime, automated composition in an inter-enterprise collaborative virtual enterprises in which semantic web/grid of collaborating partners are shared in a service registry. The service composition middleware is a utility infrastructure, which accepts user-request on demand and process the request to achieve runtime, automated service composition.

Our prototype implementation was carried out using Java technologies and some other relevant frameworks. Java 2 Standard Edition 1.5 running in Netbeans IDE version 5 was primarily used as the programming environment. The Java Agent Development Environment (JADE) (JADE, 2005) framework was used to implement all the agent's roles in the system. Java Interface for Network Infrastructure (JINI) technology (JINI, 1994) provided JavaSpaces service, which is a Java based tuple space implementation that extends LINDA coordination model entries as objects. All grid services were implemented using the Globus Toolkit 4 Java Web Service core API (Globus Alliance, 2005) and deployed into the Apache Tomcat Web Container (Apache, 1998). An open source implementation of UDDI 2.0 registry JUDDI (Apache, 2003) was used as the registry to publish grid services and IBM UDDI4J (Tidwell, 2001) was used to programmatically interact with the JUDDI. WSDL4J (WSDL4J, 2005) was used to parse WSDL descriptions during the publication process of WSDL interface description in the JUDDI. The Gateway Agent uses WSDL2Java tool (WSDL2Java, 2000) to generate proxy stubs used by GSAs to invoke the service implementation. The task execution plan is generated as an XML file, which is represented in the tuple space as a Java object. Finally, OWL-S (OWL-S, 2004) was used to describe the service profiles for grid services. Figure 5.1 shows a layered overview of MINDS container while Table 5.1 summarizes the key frameworks used in the prototype implementation and their roles.

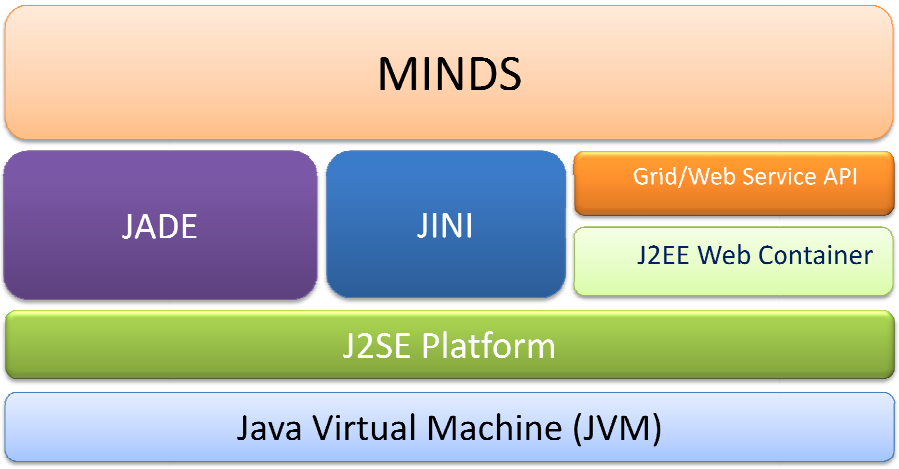

Figure 5.1: High-level layered overview of MINDS implementation

Table 5.1: Summary of platforms and frameworks used in implementation of MINDS and their roles

Platform/Framework	Description	Role in MINDS
J2SE 1.5	Java 2 Standard Edition version 1.5	The Java compiler
Netbeans 5	An IDE for Java programming	Main programming environment for components of MINDS
JADE	Java Agent Development Environment	Used to implement all the agent's roles in MINDS
JINI	Java Interface for Network Infrastructure technology	Provided JavaSpaces service - a Java based tuple space implementation based on the Linda coordination model
GT4 Java WS-Core	Globus Toolkit 4 Java Web Service Core. A Java API for implementing Grid services.	Used to implement the services used in the Itinerary planner case study as Java-based grid services
Apache Tomcat Web Container	A web container by Apache	Provides container services for deploying Web/Grid services
JUDDI	An open source Java implementation of UDDI	Used as the registry to publish grid/web

	2.0 registry	services used in the case study experiment.
UDDI4J	A Java class library for UDDI by IBM	provides an API to programmatically interact with the JUDDI
WSDL4J	The Web Services Description Language for Java Toolkit (WSDL4J) allows the creation, representation, and manipulation of WSDL documents	Used parse WSDL descriptions during the publication process of WSDL interface description in the JUDDI
WSDL2Java	Takes a WSDL document and generates fully annotated Java code	The Gateway Agent used it to generate proxy stubs needed by GSAs to invoke the service implementation
XML	eXtensible Markup Language	Task execution plan is generated as an XML file and represented in the tuple space as a Java object. Other supports files such as task execution state are also generated as XML documents.
OWL-S	Web Ontology Language for services	Used to describe the service profiles for grid/web services

5.2 A walk-through the Planit Itinerary Planner

We now focus on the Planit Itinerary planner in order to demonstrate how we address the challenges it posed for automated, runtime service composition.

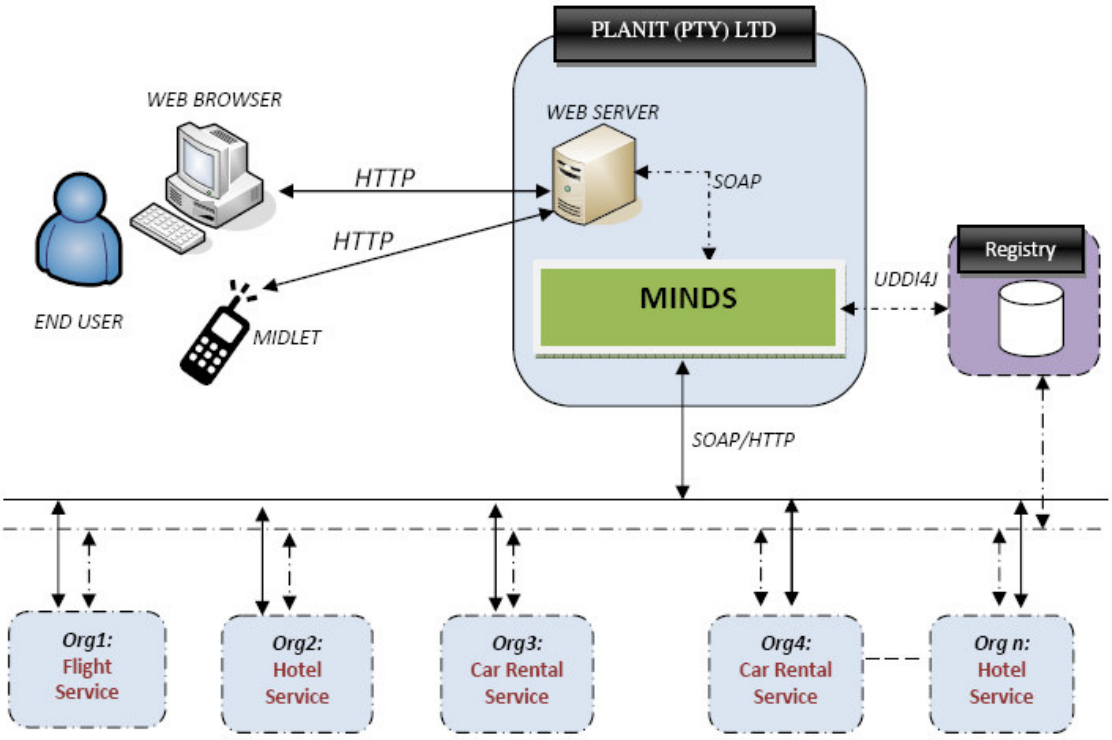

Figure 5.2: Planit scenario implementation setup

5.2.1 Overview

In Figure 5.2, we present an overview of the Planit Scenario implementation. In the experiment, we registered 20 fictitious business organizations (**org1, org2,….orgn**) in the UDDI Registry. Each of these organizations is assumed member of the South African tourism virtual organization which has published its semantic service description. The first relevant issue is the communication endpoints with the service consumers. How would end-user use the system and how would the generated

composed services be accessed? In MINDS, end-user applications or back-end systems act as service consumers. They send a semantic service request to MINDS. So, the end-user only interacts with the MINDS platform through the front-end applications which formulate semantic requests. These applications take the data from the user and build a semantic service request out of them either by using a (form) template, or by assembling a request out of given building blocks. For the Planit experimental project, we implemented the end-user interface, as both web and mobile interfaces integrated with the Planit application server through a HyperText Transfer Protocol (HTTP) request and response. The mobile environment was simulated using the Sun Wireless Toolkit 2.5 version J2ME Emulator from Microsystems. The Mobile client extended CLDC configuration and MIDP profile, which support HTTP networking. Figure 5.3 shows the sequence diagram for user request entry through the mobile interface while Figure 5.4 presents the user request entry implementation class diagram.

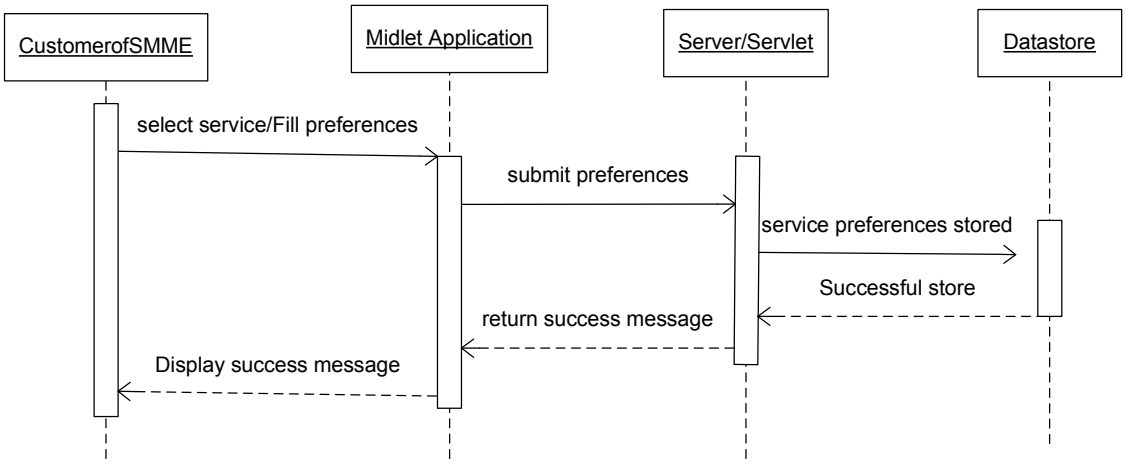

Figure 5.3: Sequence diagram for the user request submission mobile interface

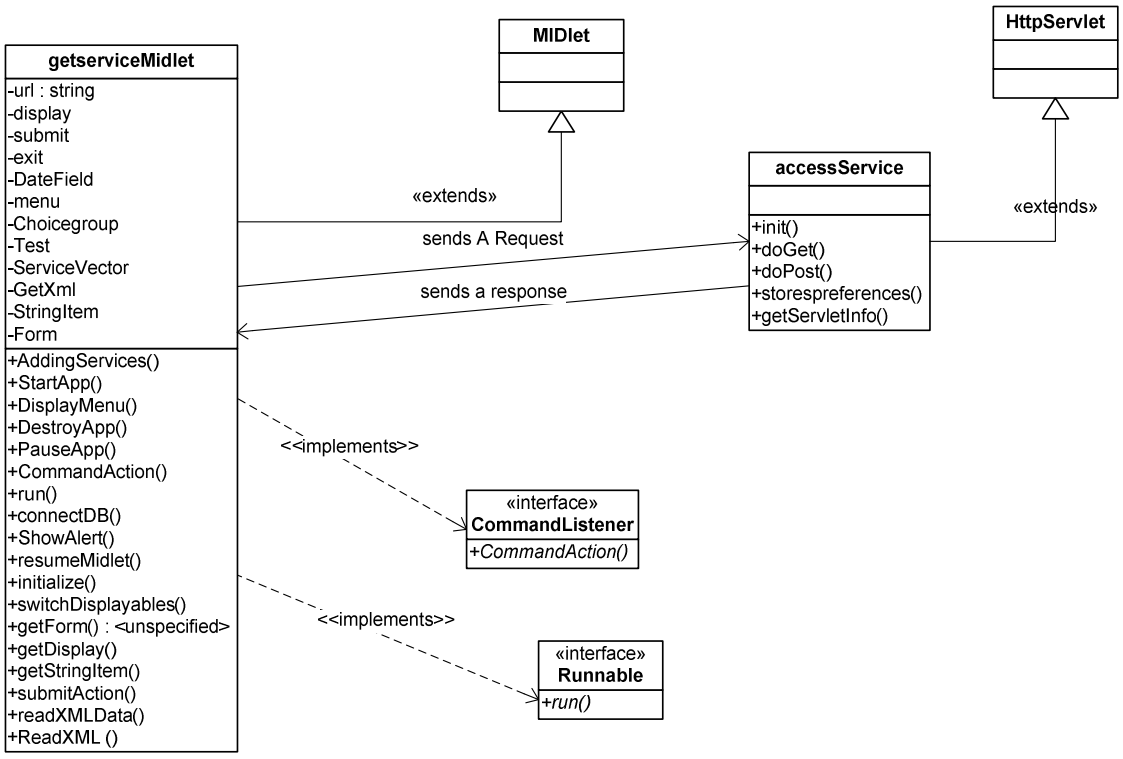

Figure 5.4: Class diagram for the user request submission mobile interface implementation

Also, with MINDS platform, human users (developers) are relieved from dealing with discovery and composition of services manually since the request is sent directly to MINDS without having to discover and compose the services at design time. Additionally, MINDS provides more reliability because it dynamically handles fault in the system without it being hard-coded into the application itself.

Communication between the Planit web server and its clients, in this case, is through HTTP[16]. The Planit Web server and internal application logic also communicate with MINDS through a web service interface via SOAP[17]. Also, invocation of composed services is done through a SOAP interface.

[16] HTTP – Hypertext Transfer Protocol
[17] SOAP – Simple Object Access Protocol

5.2.2 Illustrating major steps in MINDS Service Provisioning Life-cycle

When a customer (Joe) enters the Planit home page on its browser or through a mobile client, he gets to their landing page which is shown in Figure 5.5. Thereafter, he begins the itinerary planner by clicking on the ***"Click here to start a plan"*** button.

On getting to the request submission page, the user fills the itinerary request form which includes information on the user preferences for those services. Figure 5.6 shows a sample request submission interface used in this prototype. A mobile client version of the request submission page is shown in Figure 5.7. After submitting the request, Planit web page displays a summary of the entries made by the user for confirmation. The entry confirmation page is shown in Figure 5.8. Upon confirmation of request, user clicks on the submit request button which then translates the user request via SOAP to the MINDS platform. On getting the request, MINDS then applies its own internal logic to:

- decompose the task and represent it on the tuple space,
- performs dynamic service matchmaking to discover matching services
- dynamically synthesize compositions
- execute the task and adapt to runtime failure.

A sample partial task execution plan generated in XML format for the hotel service is shown in Figure 5.9. On completion of service composition, the result of the composition is generated and passed to the Planit web application logic for proper formatting and delivery to the customer. Results are generated as an XML file to be rendered on mobile or web interface.

Figure 5.5: Planit website landing page

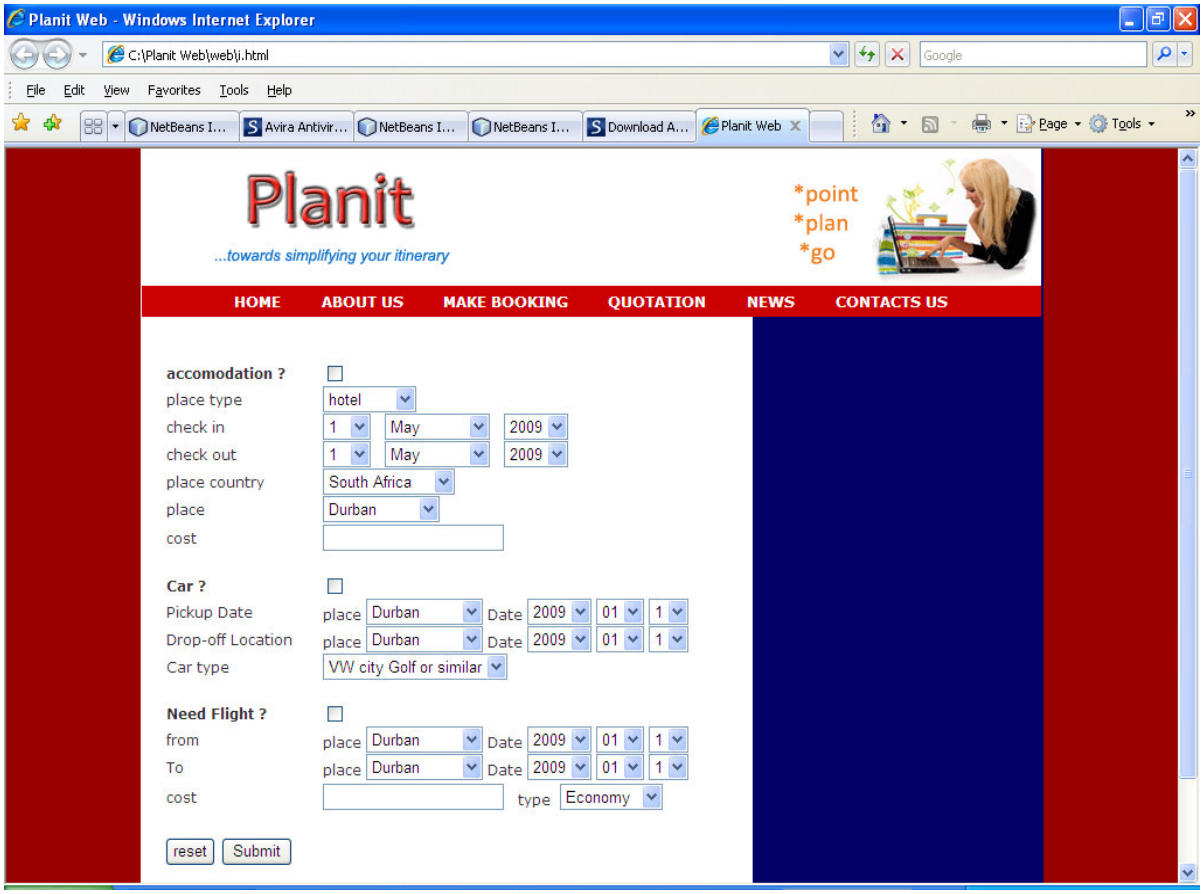

Figure 5.6: Web user request submission interface for the Plainit scenario

Figure 5.7: Mobile user request submission interface for the Plainit scenario

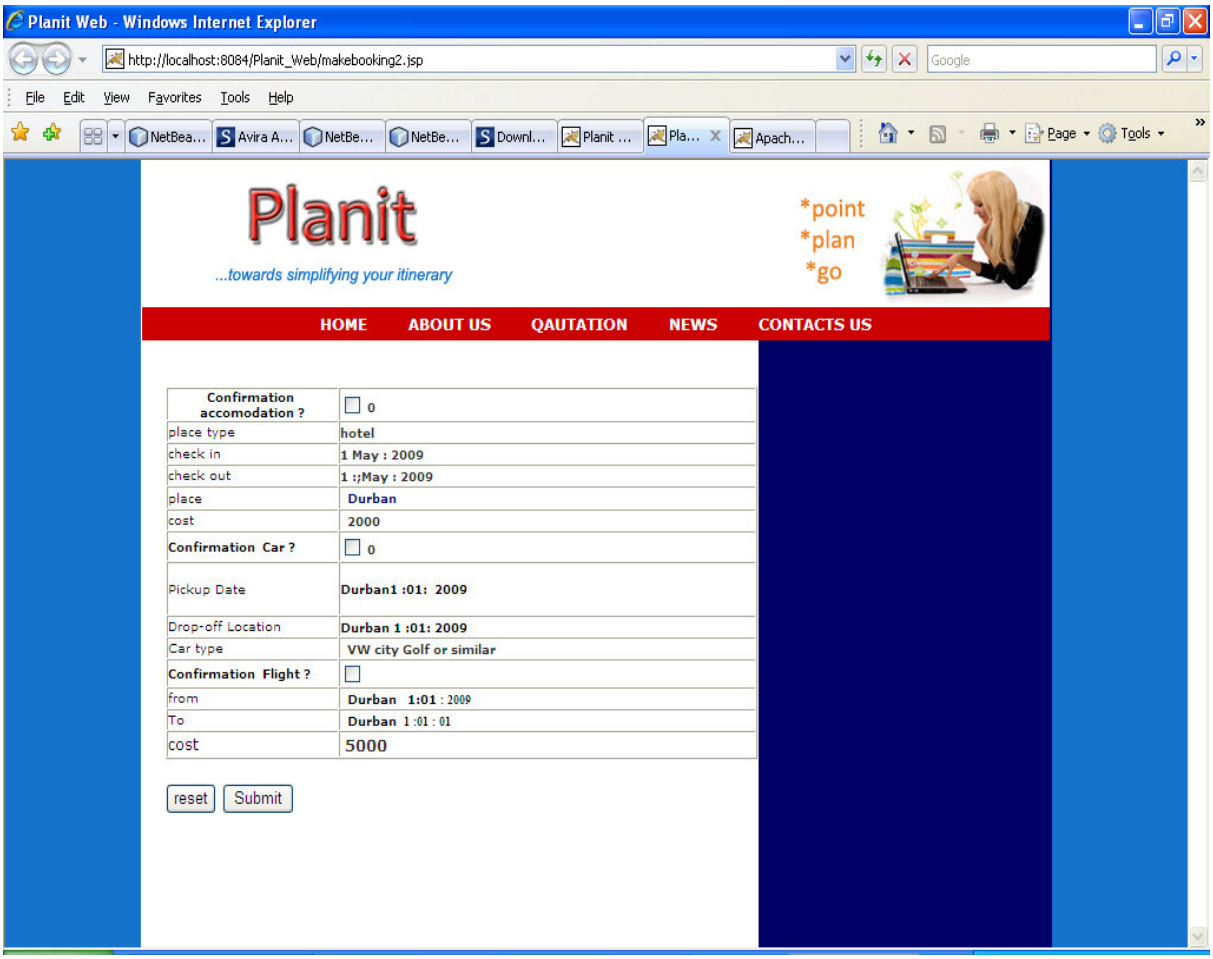

Figure 5.8: User request confirmation page for the Plainit scenario

```
<plan>

 <task id = “1324676687” name = “Hotel”>
   <dependencies>
   <dependency id = “2342543251” name = “Flight” type = “”>
      <outputList>
      <output name = “flight_ticket” type = “FlightTicket”>
     </outputList>
         </dependency>
  </dependencies>
  <execution-policy>
     <dependency-constraints availability = “all”>
          <dependency-constraints availability = “single”>
  </execution-policy>
   </task>
  </plan>
```

Figure 5.9 A sample partial task execution plan for Hotel service

5.3 Chapter Summary

In this chapter, we have presented the implementation of MINDS. We started with an overview of the Platforms and Frameworks that were used in the implementation to realize MINDS functionalities. We have also discussed our experience with the Planit itinerary planner project which illustrates a way in which the MINDS architecture can be used in a real-life setting especially for runtime, automated composition and dynamically formed virtual organization ecosystem where business services are shared on the Internet.

However, it is noteworthy that our implementation is restricted to just experimental prototype with only a handful of services deployed on a local web server and registry. While, this allows us to gain useful insights on how MINDS can be applied in practice, we need to evaluate the performance of MINDS in a controllable and repeatable experimental environment, so as to ascertain its performance tradeoffs and benefits. Towards this end, we turn in the next chapter, to an empirical evaluation of MINDS performance through a mathematical analysis and simulation experiments.

Chapter 6

Performance Analysis and Simulation of MINDS

This chapter presents an evaluation of the performance of MINDS architecture. The goal of the chapter is to investigate the performance benefits of MINDS in terms of its design decisions over some possible alternatives. Towards this goal, we employ an empirical analysis of MINDS in contrast to comparable service composition systems such as RUDDER and ACE reviewed in Chapter 2 and through extensive simulation experiments evaluates their performance using metrics such as *scalability*, *fault-tolerance* and *bandwidth cost optimization*. The choice of an empirical approach over testing the system on a real-life test-bed, is preferred because it offers us the opportunity to conduct repeatable and controllable experiments of various service composition strategies under different scenarios (e.g. varying the number of registered services, number of service providers involved in a composition, varying the failure rates for providers and resources), for quicker performance evaluation than we could have on a real-life service infrastructure test-bed.

6.1 Introduction

In order to demonstrate the effectiveness of MINDS and associated service composition strategy, its performance needs to be evaluated and compared with various alternative designs found in comparable systems such as RUDDER (Li and Parashar, 2007) and Accord Composition Engine, ACE (Agarwal et al, 2003), using different scenarios such as varying the number of registered services, number of service providers, failure rates, in order to see how each of the systems behave.

In a real life service environment, it is often practically difficult and perhaps impossible to perform evaluation of service composition middleware systems in a repeatable and controlled manner for a number of reasons. One, the environment is dynamic and services come and go at whim; therefore, controlling the services for data generation becomes practically impossible. Two, it is also impossible for an individual user or domain to control activities of other users who are in a different administrative domain because services are autonomous and can act independently. Three, a real-life testbed is often not feasible as building it requires expensive hardware and could be time-consuming. Four, even when a test-bed is available, it is often limited to a few resources and service providers making it difficult to test the system for scalability and adaptability to varying user and requirement changes.

In view of these limitations, a feasible option is the use of an empirical approach to evaluating the performance of MINDS by comparing it to some other alternative strategies found in the literature so that through mathematical analysis and extensive simulation experiments, we can investigate the following key performance questions:

i. what is the effect of increasing number of registered services on service discovery time?
ii. how does increasing the number of tasks in the composition affect service composition time?
iii. how does the failure of service provider(s) affect(s) the number of composition tasks that met deadline?
iv. how does the failure of the service monitoring agent(s) impact(s) the number of composition tasks that met deadline?
v. what is the effect of increasing the number of interacting services on the bandwidth cost?

The aim of this chapter is to provide answers to these questions through an empirical evaluation of MINDS performance. The chapter starts by defining the parameters used for the performance comparison and presents a mathematical analysis of various systems based on the parameters. This is then followed by a description of the simulation experiment and the metrics that were measured. Thereafter, an extensive

discussion of the simulation results is presented and finally, a summary of our observations and conclusions from the experiments is presented.

6.2 Basic concepts used in the Performance Model

We start the discussion in this chapter by defining the following basic abstractions and assumptions used in the performance model:

- **Service Consumer:** This is an entity that requires the capability offered by the service provider. The consumer's request is submitted through the middleware infrastructure. A service consumer is assumed to have one or more tasks to be submitted for execution.
- **Task:** A task is defined as a unit of work performed by service provider through the service provisioning engine e.g. calculate price, get quote etc. It is assumed for the purpose of this performance analysis that all tasks are atomic and not composite. It is assumed that no task is processed in its composite form. Furthermore, a task is assigned to only one service provider per time. It is only reassigned to another service provider if current service provider or agent fails. We do not consider parallel allocation of the same task to multiple providers.
- **Service Provider:** A service provider is a simple grid/web service implemented to execute a given task. We assume a service provider is only designed to execute just one atomic task at a time and not multiple tasks concurrently.
- **Service Agent:** A service agent acts as wrapper on a service provider and performs the task of monitoring task execution on the service provider, reasoning and discovering task that the service provider can participates in, and also send error notification to the coordinator in case the service provider fails. An agent can monitor one or more service providers.

The relationships between the main entities in the performance model can be depicted pictorially as shown in Figure 6.1

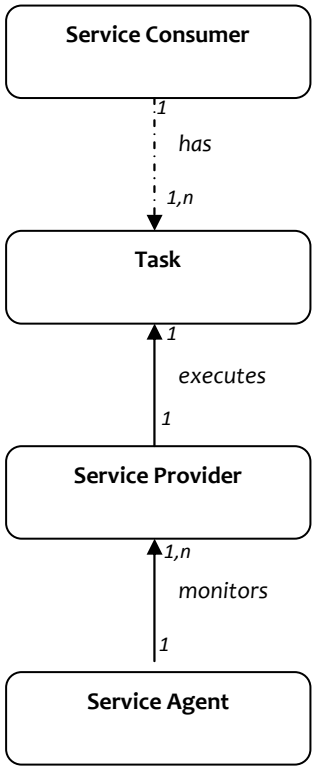

Figure 6.1: Basic concepts in the performance model and their relationships

6.3 Performance Comparison Model

For the performance comparison model, the following metrics were used:

- scalability
- fault tolerance
- network bandwidth cost optimization

We now discuss them in turn.

6.3.1 Scalability

Scalability is, basically, the ratio between performance and resources. It is important to measure scalability as a performance metric in service-oriented systems because the size of services to be selected/composed from can grow exponentially per time which can impose performance overhead on the

system. So, in measuring scalability, our desire is that as the number of services in the system increases, the system should continue to function almost as designed. Scalability, therefore, measures how well the system is able to respond when the number of service request increases. For our analysis, the relevant parameters are: (i) the time it takes for matchmaking and, (ii) the overall time it takes to complete the composition process.

6.3.1.1 Matchmaking time

The process of discovering and selecting matched services from the registry based on user request and available services' capabilities is called *matchmaking*. There are two approaches for service discovery found in the systems that we reviewed. One is the use of a *centralized agent* (or broker) found in the ACE and RUDDER systems, while the other is the used of *distributed agents* for proactively discovering matching services as employed in MINDS. We analyze these two approaches in turn:

Assume a set of services S published in the service registry, where

$$\boldsymbol{S} = \{\boldsymbol{S_i}\} \ \forall \ \boldsymbol{i} = \mathbf{1}, \mathbf{2}, \ldots . \boldsymbol{n} \qquad (6.1)$$

The objective is to find a service $\boldsymbol{S_i}$ from a pool of services $\boldsymbol{S}$ in the registry that is semantically equivalent to $\mathbf{S_{req}}$. i.e.

$$\boldsymbol{S_i} \equiv \boldsymbol{S_{req}}, \qquad \forall \boldsymbol{S_i} \in \boldsymbol{S} \qquad (6.2)$$

(A) Case 1 - Centralized broker agent:

When a centralized broker agent is used, the number of searches that has to be performed to discover $\mathbf{S_{req}}$ is naturally unknown. Thus, if we assume a random number of searches **p** in **n** is performed before $\mathbf{S_{req}}$ is discovered. And, assuming a uniform search time ($\mathbf{S_t}$) per service in the registry, then total time required to discover $\mathbf{S_{req}}$ for each task in the composition is,

$$\boldsymbol{t_i} = \boldsymbol{p} * \boldsymbol{S_t} \qquad (6.3)$$

Total discovery time for centralized broker agent ($\mathbf{t_{dc}}$) is

$$t_{dc} = \sum_{i=1}^{m} t_i \qquad (6.4)$$

where m = no of tasks in the composition.

(B) Case 2: Distributed Matchmaking agents

For distributed matchmaking agents, they constantly listen to the coordination space to see the advertisement of a new task that semantically match their service profile and therefore proactively announce interest. Therefore, if we assume a random periodic amount of time ($\mathbf{lt_i}$) in which an $\mathbf{agent_i}$ listens to the tuple space for new task advertisement it can participate in, then total time it takes to discover all agents to participate in a composition would be equivalent to maximum of $\mathbf{lt_i}$

Thus, total matchmaking time for distributed agents is given by

$$t_{dd} = \mathbf{max}_{i=1\ to\ m}(lt_i) \qquad (6.5)$$

6.3.1.2 Service Composition Time (SCT)

We define *service composition time* as the time between when a task is submitted for composition and when the task is completely executed.

Generally, service composition time (**SCT**) involves four components:

(i) matchmaking time ($\mathbf{t_s}$)

(ii) negotiation time to select the best service ($\mathbf{t_n}$)

(iii) time to generate task execution plan ($\mathbf{t_g}$), and

(iv) time to execute task ($\mathbf{t_e}$)

So, service composition time can be viewed as a quadruple,

$$\boldsymbol{SCT = f(t_s, t_n, t_g, t_e)} \quad (6.6)$$

Thus,

$$\boldsymbol{SCT = t_s + t_n + t_g + t_e} \quad (6.7)$$

However, negotiation is not compulsory in all service matchmaking and in order to simplify our analysis, we assume a constant negotiation time for all the schemes. Also, since all task execution plans are generated by a single agent in all the schemes we reviewed, we assume this to be a constant time. Moreover, matchmaking time had already been formulated in section 6.3.1.1, we then focus on analyzing the task execution time (t_e).

Similarly, the execution model used in any service composition system can be centralized or decentralized. For the systems that are based on a workflow, the execution model is orchestration which is based on a centralized coordinator. ACE uses a centralized execution model while RUDDER and MINDS are based on a decentralized execution.

If a centralized execution engine is used in the scheme, then $\mathbf{t_e}$ is the time to execute all tasks. Therefore,

$$\boldsymbol{t_e = f(m)} \quad (6.8)$$

where, $\mathbf{m}$ = *no of tasks in the composition,*

If we assume a random period of time $\boldsymbol{t_i}$ to execute each task $\boldsymbol{i}$ in $\mathbf{m}$, then $\boldsymbol{t_{ec}}$ is the time to execute all task with centralized execution model. Therefore,

$$\boldsymbol{t_{ec} = \sum_{i=1}^{m} t_i} \quad (6.9)$$

However, when a decentralized execution engine is employed, then the time it takes to complete execution would be the time it takes to execute the longest tasks in the composition since all the tasks are scheduled for execution in parallel. And, where there are dependencies, then the execution time becomes a summation of task execution time of the provider and the one it is depending on.

Therefore, $\boldsymbol{t_{ed}}$, the time to execute all tasks in decentralized execution model is given by:

$$\boldsymbol{t_{ed} = Max\ (t_i),} \qquad \forall\ \boldsymbol{i = 1, \dots, m} \tag{6.10}$$

Furthermore, the service composition process for the three systems considered in this performance analysis analysis, involves three key design strategies:

(i) centralized matchmaker and centralized execution engine

(ii) centralized matchmaker and decentralized execution engine

(iii) decentralized matchmaker and decentralized execution engine

We, therefore, discuss our analysis of these three cases in turn.

A. Case 1: Centralized Matchmaker + Centralized Execution Engine

Some service composition systems are based on a centralized matchmaking agent and a centralized execution engine. This is the case with ACE. Thus, the service composition time in this case, SCT_{cc} is given by:

$$\boldsymbol{SCT_{cc} = t_{sc} + t_{ec}} \tag{6.11}$$

Where, t_{sc} = time for centralized matchmaking

and t_{ec} = execution time for centralized

B. Case 2: Centralized Matchmaker + Distributed Execution Engine

In this case, the system is based on a centralized matchmaking agent and a distributed execution engine. This is the case with the RUDDER framework. Thus, the service composition time in this case, SCT_{cd} is given by:

$$\boldsymbol{SCT_{cd} = t_{sc} + t_{ed}} \quad (6.12)$$

Where, t_{sc} = time for centralized matchmaking

t_{ed} = execution time for decentralized

C. Case 3: Distributed Matchmaker + Distributed Execution Engine

In this case, the system is based on a distributed matchmaking agent and a distributed execution engine. This is the cased with MINDS. Thus, the service composition time in this case, SCT_{dd} is given by:

$$\boldsymbol{SCT_{dd} = t_{sd} + t_{ed}} \quad (6.13)$$

Where, t_{sd} = time for decentralized matchmaking

t_{ed} = execution time for decentralized

Next, we consider the second of the three metrics in our performance comparison model, which is fault tolerance.

6.3.2 Fault Tolerance

Most distributed systems experience partial failure. A **partial failure** occurs when one component in the system fails. The failure may affect the proper operation of some components; while at the same time leave other components totally unaffected. Therefore, an important goal in distributed service oriented systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance. In particular, whenever a failure occurs, the system

should continue to operate in an acceptable way, while repairs are being made. Being fault-tolerant is synonymous to achieving dependability in distributed systems, metrics of which are availability, reliability, safety and maintainability (Kopetz and Verissimo, 1993).

In this analysis, our interest in fault-tolerance is to measure how the system is designed to be able to provide its services in the presence of faults.

There are two types of failure that can occur in an agent-based distributed service oriented system:

(i) failure of the Service Agent
(ii) failure of the Service Provider

The descriptions of these faults are summarized in Table 6.1.

Table 6.1: Types of faults and their descriptions

Types of Failure	Description
Service Agent Failure	This happen when the agent that monitors the service execution (i.e execution engine) fails but the underlying service provider is still active and operational.
Service Provider Failure	This is the case when the service provider fails but the agent that monitors the service execution (i.e. execution engine) is still active and operational.

When there is failure of either a service agent or provider, the failure affects the completion of tasks that are assigned to that provider. Subsequently, the

failure might impact the number of tasks that is completed within deadline in the whole system if the service provider or agent did not recover from the failure on time. However, before we go into a detailed discussion of these two failure scenarios on the system, it is important to note that a correct analysis of the impact of these failures cannot be carried out without information on some standard task characteristics and scheduler measurements that serve as input to the performance analysis model. Table 6.2 presents the task characteristics and scheduler measurements that serve as input to our performance analysis model.

Table 6.2: Some standard task characteristics and scheduler measurements that serve as input to our performance analysis model

Metric		**Description**
Name	**Abbr.**	
Task Assign Time	$tass_i$	Time at which a task ***i*** was assigned to service provider , $\forall\, i \in m$ (m= no of tasks to be executed)
Latest Finish Time	lft_i	The time at which the service provider should have completed task ***i*** for the task to meet deadline
Estimated execution time	$texec_i$	Estimate of how long it takes to execute task **i**
Mean time before error	$terr_i$	The time at which service provider executing task ***i*** **or** agent in charge of task i fails
Time Spent	$tspt_i$	The time that service provider has spent on task **i** before the error
Time to Discover	$tdis_i$	The length of time from when there was error in service execution to the time that the error was discovered

Mean time to Recover	$MTTR_i$	The mean time it takes for service provider or agent handling task i to recover from its fault
Reassign Time	$treassign_i$	Time it take to reassign task i to another provider when failure occur
Task used time	tct_i	A measure of how long time has been used by both the provider and fault recovery mechanism on task i, this is measured as: $tct_i = tspt_i + MTTR_i$
Monitoring Time	*Tslice*	A quantum slice of time that every Service Provider gets monitored by the Service Agent

We now present our analysis based on the two scenarios described earlier.

6.3.2.1 Scenario 1: Service Agent Failure

Assume a task **i** is assigned to a service provider at time $\boldsymbol{tass_i}$, $\forall\, \boldsymbol{i} \in \boldsymbol{m}$ **.** We further assume that all task execution has a ***latest finish time*** ($\boldsymbol{lft_i, i = 1, \dots, m}$).

The ***latest finish time*** is the time when a service provider should have completed the task assigned to it and the results returned through the service agent.

Therefore, if the service agent fails at an instance of time denoted by ***error time*** $terr_i$, we measure the ***task used time*** tct_i as the sum of the time the

service provider has spent executing the task, $tspt_i$, and the failure recovery time $MTTR_i$ of the service agent.

That is:

$$\boldsymbol{tct_i = tspt_i + MTTR_i} \quad (6.14)$$

However,

$$\boldsymbol{tspt_i = terr_i - tass_i} \quad (6.15)$$

Therefore, if the task completion time (tct_i) is greater than the latest finish time for task i (lft_i) then the task could no longer meet the deadline when it is retained with the current service provider. Therefore, the service provider is assumed to have failed since it could not deliver its result through the service agent. The task assigned to that provider is then **reassigned**. Otherwise, the task i would meet the deadline.

It is important to then note here that, in a case where a centralized coordinator agent is used for execution monitoring, when the global agent fails, there would be more tasks that would be assumed failed because they could not transmit their output to other dependent services.

6.3.2.2 Scenario 2: Service Provider Failure

When the Service Provider fails, it means the task could not be executed on it any longer and probably the task would need to be reassigned to another node and the service agent terminated. A very important factor in this case is how long it takes to discover that the service provider has failed and reassign its task. This is, however, dependent on the workload of the task execution monitoring entity.

In a case where there is just one central entity (agent) that monitors all the Service Providers, if we assume that monitoring time is sliced and each Service Provider only get monitored at an assigned uniform quantum of time denoted

by ***tslice;*** then a failed service provider would have to wait for an interval of time denoted by

$$\boldsymbol{tdiscover_{i(centralized)} = random\,(m) * tslice} \qquad (6.16)$$

This means the waiting time is a function of a random value of the relative positioning of the monitored provider in the queue of the service agent and the time allocated to each provider.

However, in a decentralized case, the service agent directly monitors a service provider, this would then result in a reduction in the failure discovery time and therefore tasks can easily be reassigned. The time it takes to detect fault would, therefore, be a maximum of the periodic interval at which the agent carry out monitoring (assumed to be ***tslice***)

Therefore the time for discovery in a decentralized execution is:

$$\boldsymbol{tdiscover_{i(decentralized)} = p, \quad where, \quad p \leq tslice} \qquad (6.17)$$

It then follows, from the foregoing, that if the task completion time (tct_i) is greater than the latest finish time for task i (lft_i) or if the estimate of how long it takes to execute task **i** ($\mathbf{texec_i}$) plus the time to discover that task i is in error ($tdiscover_i$) is greater than the latest finish time for task i (lft_i), then the task could no longer meet the deadline when it is retained with the current service provider. Therefore, the service provider is assumed to have failed since it could not deliver its result through the service agent. The task assigned to that provider is then **reassigned**. Otherwise, the task i would meet deadline

6.3.3 Network Bandwidth Cost Optimization

A very important resource during service composition is the data that is transmitted during the execution of tasks. If the location and transfer of data is not well managed by the system, it would constitute performance overhead on the network bandwidth. Therefore, a way of improving the performance of the system is to optimize network bandwidth by reducing the amount of time taken for data transmission on the network.

Although, there are control and data flows during any composition process, here, our interest is limited to bandwidth overhead caused by data flow.

In service composition system that requires transfer of large chunks of data as found in many scientific and business applications, passing large quantities of intermediate data through a centralized orchestration engine results in unnecessary data transfer, wasted bandwidth, engine overload, and diminishing performance of service composition system (Barker et al, 2009b).

As an illustration, let us assume a simple case of data flow among five constituent services (see Figure 6.2) namely *service 1, service 2, service 3, service 4*, and *service 5* as presented in Barker et al, 2009b. Data input to *service 4* (600MB) is generated from data outputs from *service 1* (200MB), *service 2* (200MB) and *service 3* (200MB). Also, *service 4* produces as data output 100MB which serves as input to *service 5*. Finally, *service 5* generates 120MB of data which is then the final output to the user. As shown in Figure 6.2a if involved services are orchestrated, the data flow pass through a centralized workflow engine. Therefore, in order to enact this scenario, the total data flow through the system will be 1520 MB.

However, if a decentralized model is adopted such as in choreography or collaboration, the output of the service invocation can be passed directly to where it is required as input to the next service. This is represented in Figure 6b. Since data do not have to pass through a centralized engine, the

decentralized approach involves a total data transfer of 820 MB, assuming no further data transformation takes place and the final output needs to be sent to the end user. Therefore, using a decentralized model, the total data transfer needed to enact the workflow is 700MB less when compared to orchestration.

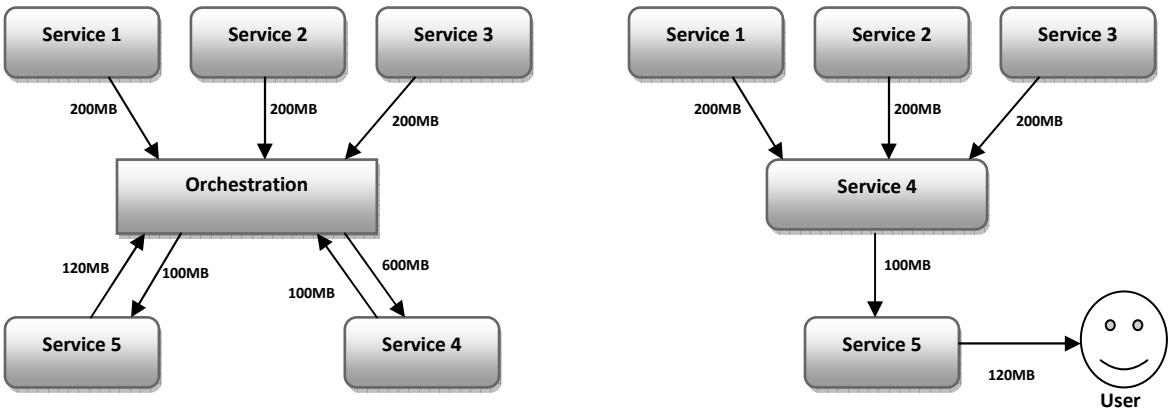

Figure 6.2: Data flow scenario (a) centralized orchestration engine

(b) decentralized execution (*Adapted from*: Barker et al, 2009b)

To develop a generic analytical model for the behavior of both centralized and decentralized execution model in terms of bandwidth cost optimization, we use a weighted graph to model the data flow operations of both systems. The vertices of the graph represent the *services* and the edges represent the *communication link* between services (Grossman, 2002). The weights of the edges represent the *bandwidth costs* of data transferred between the services. We then develop a *dependency graph* which shows us the flow of input and output (I/O) data between the various services participating in the composition (Omer and Schill, 2009). For simplicity, we assume a simple dependency relation where a service does not have more than one dependent. The weighted graph could be represented computationally using a matrix similar to an adjacency matrix, but now putting the weight on the edge from

vertex v_i to vertex v_j, rather than put the number of edges, in row *i* and column *j*.

Therefore, if we let G be a weighted graph with vertices $v_1, v_2, v_3, \ldots, v_n$, in which the weight represent the communication cost. The *weight matrix* of G is the *n x n* matrix for which the entry w_{ij} in the *i*th row and the *j*th column is given by the rule:

$$w_{ij} = \begin{cases} 0 \; if \; i = j \\ \text{the weight of the edge from } i \text{ to } j \text{ if } v_i \text{ and } v_j \text{ are adjacent} \\ \infty \text{ if } i \neq j \text{ and } v_i \text{ and } v_j \text{ are not adjacent} \end{cases} \quad (6.18)$$

Similarly, the dependency graph could be represented by a dependency matrix which shows the input-output (I/O) dependency between services. This matrix will be a square matrix (nXn) where *n* equals available services to form the composite service. Each row and column represents candidate services for the composite web service (WS_i). If a service on *i*th column is dependent on a service on the *j*th row the *C*ij value of the matrix will be 1 otherwise it will be zero.

Thus, if we let the composite service to be created require *n* services, WS_1, WS_2 ,…., WS_n. Then the dependency matrix (DM) can be defined as follows:

$$\text{DM} = \begin{bmatrix} C11 & \cdots & C1n \\ \vdots & \ddots & \vdots \\ Cn1 & \cdots & Cnn \end{bmatrix} \; where \; C_{ij} = \begin{cases} 1 \; if \; WS_i \; is \; dependent \; on \; WS_j \\ 0 \; otherwise \end{cases} \quad (6.19)$$

Hence, using the *dependency matrix* and the *weighted matrix* it is possible to determine the flow of data and calculate the cost of data flow in both

centralized and decentralized execution model using the following steps (detail algorithm is as shown in Figure 6.3):

1. identify explicit direct dependencies from input and output parameters of services in the registry and construct a dependency matrix (DM)
2. input a connected weighted graph G, with vertex set V(G) = {v1, v2,…, vn} and weighted edge set E(G) = {e1, e2,…, em}
3. from the dependency matrix, calculate sum of values on each rows (S_Rowi) and columns (S_Coli) in DM
4. using the sum values on DM, determine the index of the starting and ending node in the composite service
5 determine the cost of data transfer from start node to end node
6. output the data transfer cost for both centralized and decentralized scheme.

Algorithm: ***Calculate bandwidth costs***

Step 1: Identify explicit direct dependencies from input and output parameters of a services and construct a ***dependency matrix*** (**DM** = $\mathbf{C}_{ij}$);

Step 2: Input a connected ***weighted graph G***, with vertex set **V(G)** = {$v_1, v_2, \ldots, v_n$} and weighted edge set **E(G)** = {$e_1, e_2, \ldots, e_m$} (**n** = *no of services in the composition;* ***m*** = *no of edges*)

Step 3: From the dependency matrix, calculate sum of values on each rows (S_Row_i) and columns (S_Col_i) in DM, where i=1…n (note: maximum sum value is 1 since there is one dependent service);

Step 4: Using the sum values on DM, determine the index of the starting and ending node in the composite service as follows:

```
procedure findNode
    loop until n
       if S_Col_k == 0 then Start = k;  //determine index of start node
    loop continue
    loop until n
        if S_Row_j == 0 then End = j;  // determine the index of end node
     loop continue
    return Start, End;
end procedure
```

Step 5: Determine the cost of data transfer from v_{Start} to v_{End} as follows:

```
procedure calculateCost
    W_central = 0, W_decentral = 0; //initialize variable to store cost for each scheme
    c_node = Start;    //c_node is current node marker
        loop until n          //index <- current loop index value
            if (C_c_node,, index == 1) then  //is there connection between c_node and index
                if (system == decentralized) then
                        W_decentral += e_c_node,index;   //data sent directly to next node
                         c_node = p;
                end if
                if (system == centralized) then
                    if (c_node != End) then
                        W_central += 2 * e_c_node,p   //data is first sent to the orchestration
                                                      // engine, hence doubled in value here
                         c_node = p;
                    else
                        W_central += e_c_node,p
                    end if
                  end if
            end if
          loop continue
end procedure
```

Figure 6.3: Algorithm to calculate the bandwidth cost for centralized and decentralized schemes

This section addresses the simulation program developed based on the performance analysis presented above.

6.4 Simulation Experiments, Results and Discussions

6.4.1 The Simulation setup

The simulator for the test cases was developed using Java Standard Edition 5 running in Netbeans IDE 6.5. We carried out various simulation experiments that investigated the following:

(1) matchmaking time as the number of registered services increase;
(2) service composition time as the number of tasks in composition increase;
(3) number of tasks in composition that meet deadline when the service agent(s) fail;
(4) number of tasks in composition that meet deadline when the service providers fail;
(5) bandwidth cost as the number of communicating service providers increase.

The parameters used in the simulation experiment and their default values are given in Table 6.3.

6.4.2 Results and Discussions

6.4.2.1 Matchmaking Time as the Number of Registered Services increase

The aim of this experiment was to investigate the scalability of the systems in terms of the time it takes for service matchmaking when the number of registered services increases. The result obtained from this simulation experiment is as shown in Figure 6.4, while Table 6.4 shows the simulation data.

Table 6.3: Parameters and their default values for the simulation experiments

Parameter type	Value
Max discovery time in distributed (secs)	240
Search time per service (secs)	10
No of Failed Service Providers	[2,40]
Latest Finish Time (secs)	130% original time allotted
Processor slice time per Provider (secs)	5
No of Services	[5,105]
No of tasks	[1, 110]
Average task execution time (secs)	[1,300]
Data transmission cost between services (for Weighted Matrix)	[5,50 Rand]

Table 6.4: Simulation Data for Matchmaking Time vs No of Registered Services

No of Registered Services	**Time Centralized (Secs)**	**Time Decentralized (Secs)**
10	140	213
15	280	210
20	370	238
25	410	236
30	320	240
35	440	238
40	580	232
45	660	218
50	580	237
55	460	239
60	610	235
65	1560	238
70	1090	239
75	1080	240
80	1130	240
85	1130	240
90	1100	240
95	1390	234
100	740	239
105	1200	239

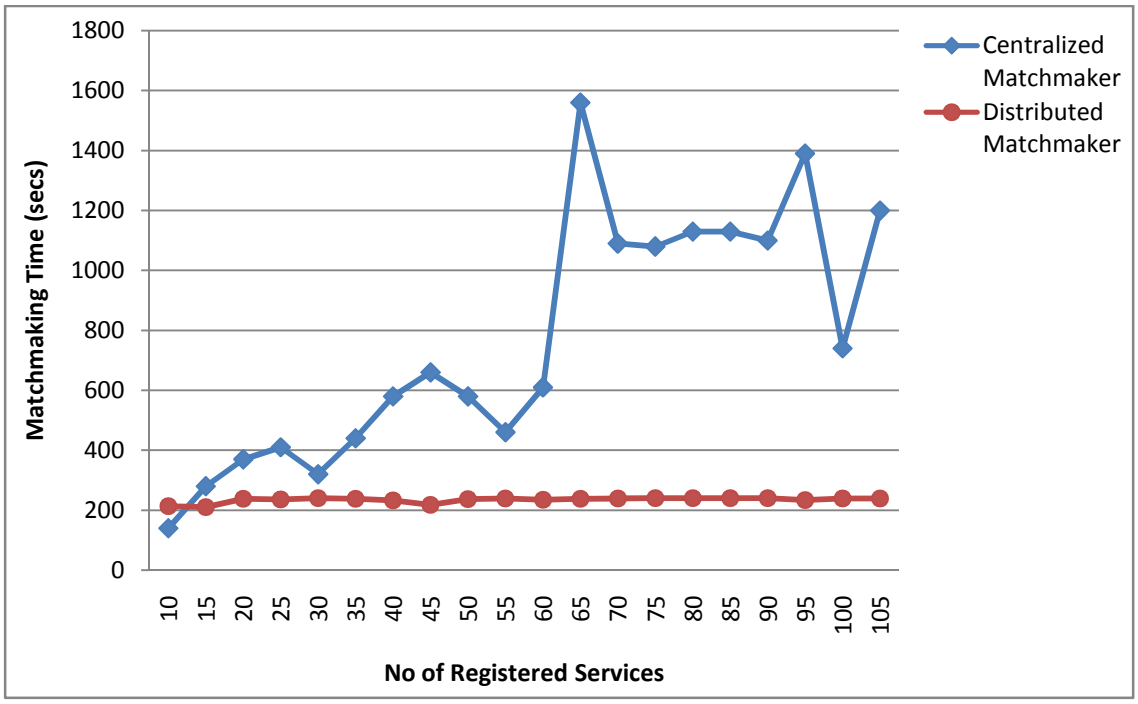

Figure 6.4: Matchmaking Time vs No of Registered Services for Centralized and Decentralized Agents

From the result, it could be observed that at lower number of registered services, the traditional centralized matchmaker approach outperforms the distributed matchmaking approach employed in this study in terms of matchmaking time. However, as the number of registered services increase, the time required to perform matchmaking in a centralized system began to increase gradually while the one for the decentralized system is still relatively constant. For example, when 10 services were registered it took 140 secs to discover and perform matchmaking in the centralized approach as against 213 secs in the distributed approach. But when 75 services were registered, it took 1080 secs to perform matchmaking with centralized technique while it took 240 secs for the distributed matchmaking approach. It could then be seen that there is really no significant benefit of decentralized strategy over the traditional centralized matchmaking strategy when dealing with a handful of services in the composition. However, as the number of registered services

increases, decentralized matchmaking strategy becomes preferable because each service agents individually monitor service advertisement and listen for arrival of new service matching their interest. This, therefore, cumulatively results in reduction of time to discover and select services required for the entire composition. Therefore, we can then conclude that as the number of registered services increases, the scalability of the system becomes a critical issue in terms of service matchmaking time. However, as observed in this experiment, decentralized architecture employed in MINDS scale better than systems that employ a centralized matchmaker or broker such as ACE and RUDDER.

6.4.2.2 Service Composition Time against Number of Task in Composition

Our aim in this section is to investigate the scalability of the systems in terms of the time in takes to complete service composition as the number of task in the composition increases. As already discussed in the analysis section (Section 6.3), there are three design strategies found in the systems for our consideration. One, *Centralized Matchmaker and Centralized Execution Engine (CMCE)* employed in ACE. Two, *Centralized Matchmaker and Decentralized Execution Engine* (CMDE) employed in RUDDER. Three, *Decentralized Matchmaker and Decentralized Execution Engine (DMDE)* employed in MINDS. Three experiments were carried out here. Figures 6.5-6.7 show the results of the three experiments conducted with number of services fixed at 30, 50 and 100 respectively while Tables 6.5-6.7 show their respective simulation data. The reason for conducting the experiment with different number of registered services is to see the impact of different number of registered services on the service composition as we vary the number of tasks in the composition.

From the simulation results, the service composition time for all the three strategies (CMCE, CMDE and DMDE) were nearly the same when the

number of tasks in the composition is small. However, as we increase the number of tasks in the composition, the service composition time for both CMCE and CMDE strategies increases at a very rapid rate while the DMDE strategy was almost constant as number of tasks in the composition increases. For example, as shown in Figure 6.5, DMDE did not show a performance benefit in terms of scalability when number of task in the composition was between 1 and 3, but as the number of tasks increases to 4 and above, it shows performance gain in terms of lower service composition time than the CMCE and CMDE strategies. Also, it could be observed that as the number of services increases from 30, 50 and 100, the gap between the CMCE and CMDE shrinks and they nearly require the same time for composition, while DMDE maintains a relatively low amount of service composition time. One possible reason for the shrinking gap between CMCE and CMDE as the number of services is increased is that the effect of the matchmaking time overshadows the performance gain of CMDE over CMCE in terms of execution time.

Table 6.5: Simulation Data for Service Composition Time (in secs) against No of Task in Composition at Service at No of Service = 30

# TASKS	CMCE	CMDE	DMDE
1	185	186	320
2	578	530	285
3	549	572	500
4	1069	894	432
5	1685	1504	590
6	1672	1223	431
7	1695	1250	354
8	1782	1436	543
9	2317	1650	538
10	2924	2336	454
11	3321	2386	401
12	2939	2106	445
13	3298	2441	536
14	3478	2455	423
15	3612	2638	544
16	4090	2635	472
17	3457	2379	536
18	4616	3491	561
19	3991	2411	527
20	4735	2947	543

Table 6.6: Simulation Data for Service Composition Time (in secs) against No of Task in Composition at Service at No of Services = 50

# TASKS	CMCE	CMDE	DMDE
1	286	187	274
2	1088	1153	483
3	1562	1427	406
4	1650	1534	434
5	2060	1710	343
6	1901	1658	428
7	2505	2244	479
8	2937	2618	534
9	2966	2510	575
10	3796	3151	499
11	4073	3312	482
12	4699	3370	450
13	4652	3718	547
14	4791	3850	446
15	4715	3360	520
16	6251	5498	537
17	6238	4854	455
18	6172	4695	523
19	6864	5643	590
20	6824	5385	406

Table 6.7: Simulation Data for Service Composition Time (in secs) against No of Task in Composition at Service at No of Services = 100

# TASKS	CMCE	CMDE	DMDE
1	831	939	357
2	1759	1642	260
3	2267	2174	399
4	1715	1718	418
5	3384	3194	547
6	4415	3987	507
7	4032	3447	443
8	3566	3028	487
9	4627	4298	466
10	5494	4905	481
11	7104	6043	533
12	7118	6546	464
13	7747	6942	461
14	9209	8265	532
15	10310	9196	504
16	9041	7843	473
17	10650	9329	518
18	9917	8492	551
19	11240	9705	574
20	11507	9799	499

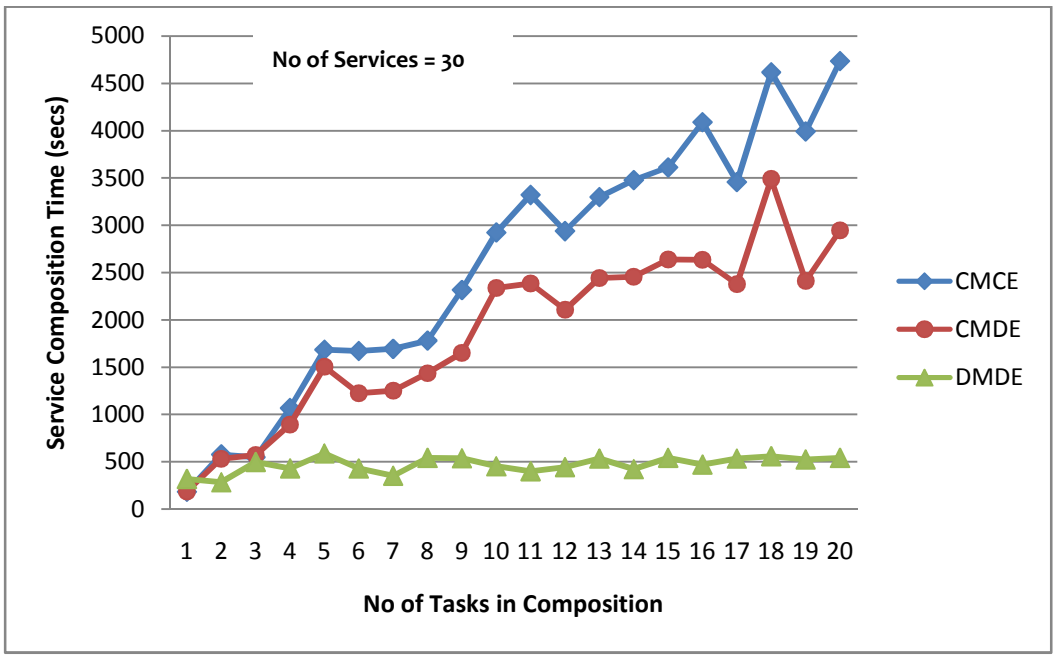

Figure 6.5: Service Composition Time vs No of Tasks in Composition at No of Registered Services = 30

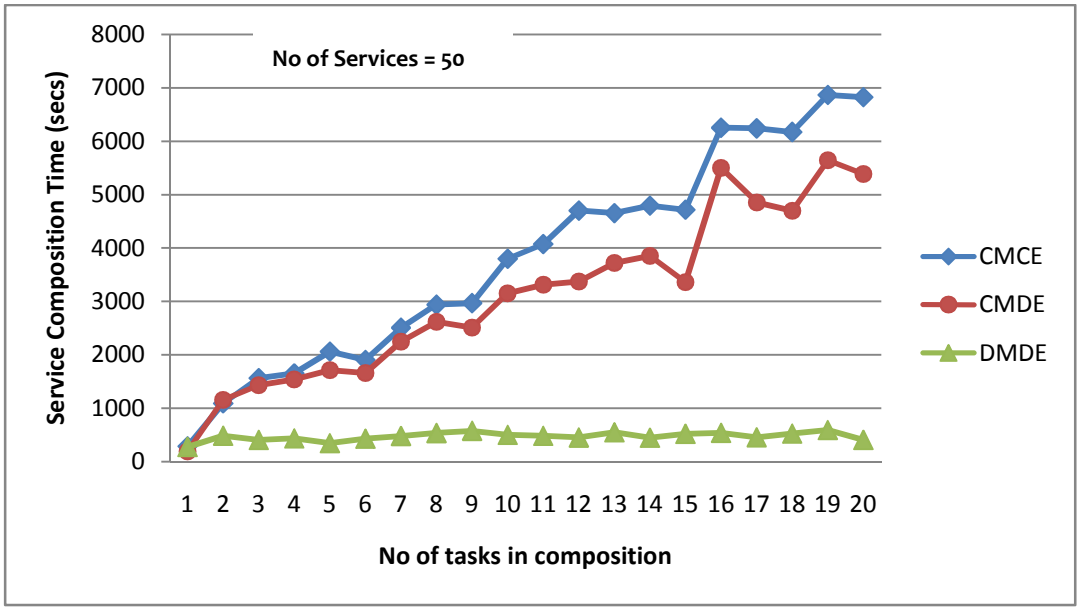

Figure 6.6 Service Composition Time vs No of Tasks in Composition at No of Registered Services = 50

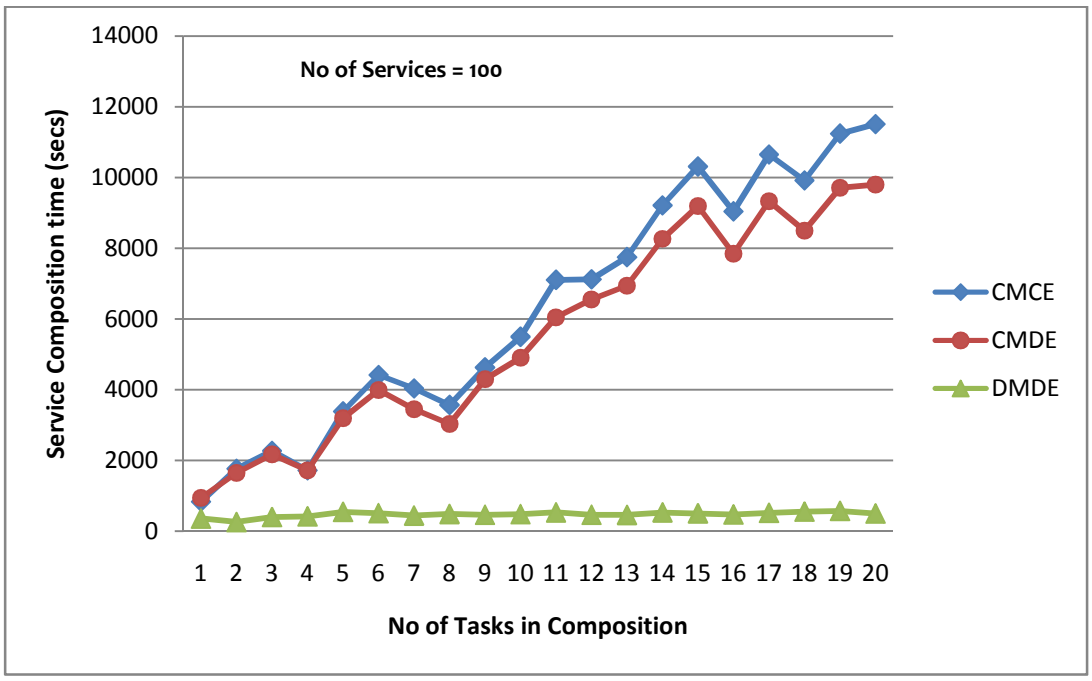

Figure 6.7. Service Composition Time vs No of Tasks in Composition at No of Registered Services = 100

6.4.2.3 Number of Tasks in Composition that meet deadline when the Service Agent(s) fail

In this and the next sub-section, we investigate the fault tolerance of systems as presented in Section 6.3.2. In this case, we monitor the number of tasks that are scheduled for composition that met deadline when there are partial failures of service agents as the number of tasks in the composition increases. The result obtained from the simulation is shown in Figure 6.8, while Table 6.8 shows the simulation data for the experiment.

From the simulation, it could be observed that as the number of tasks in the composition increases, the rate of jobs that met deadline in the Centralized agent approach employed in ACE is lower than that of decentralized service

agents employed in MINDS and RUDDER. The reason for this is that a centralized service agent introduces a single locus of failure which can jeopardize the successful completion of a lot of time-critical tasks, whereas in distributed service agent, the failure of a service agent does not impair on the functioning of the overall system.

Table 6.9: Simulation Data for No of Tasks that met deadline when Service Provider(s) fail

# Tasks	# Meet Deadline (Centralized)	# Meet Deadline (Decentralized)
10	3	7
15	6	11
20	5	13
25	11	16
30	16	22
35	14	27
40	15	26
45	14	30
50	15	34
55	16	40
60	25	41
65	25	47
70	32	49
75	34	55
80	34	55
85	35	64
90	35	61
95	41	66
100	36	70
105	50	71

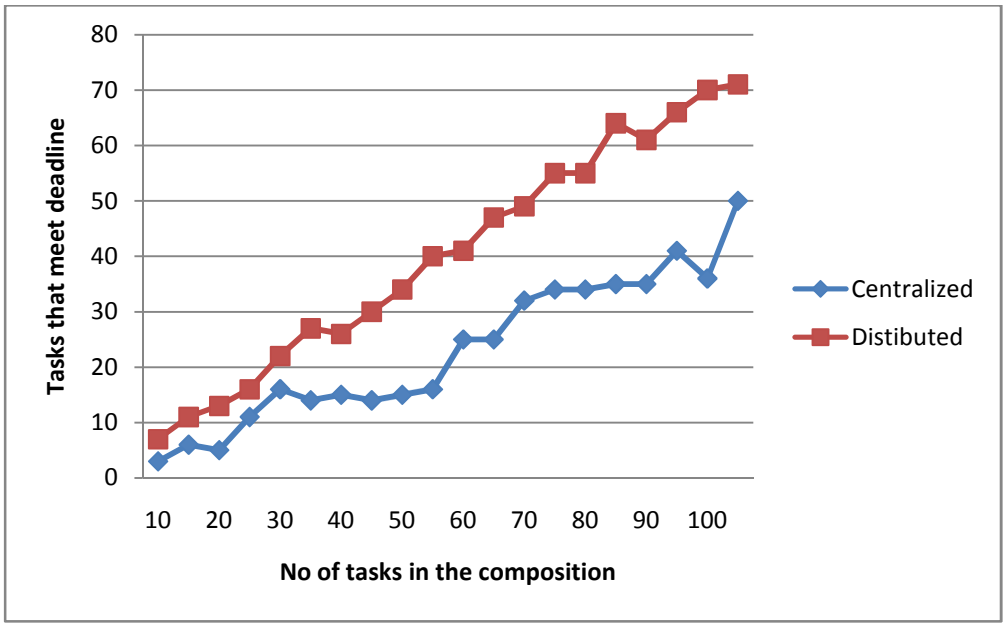

Figure 6.8. No of Tasks that met deadline vs No of Tasks in the Composition

6.4.2.4 Number of Tasks in Composition that meet deadline when the Service Provider(s) fail

As in the sub-section 6.4.2.3, our aim here is to monitor the number of tasks that are scheduled for composition that met deadline as the number of service providers that failed increases. The result obtained from the simulation is shown in Figure 6.8, while Table 6.9 shows the simulation data for the experiment.

From the simulation, it could be observed that as the number of service providers that failed increases, the number of tasks that met deadline in centralized service agent scheme (as in ACE) is lower than that of decentralized service agents (as in the case of MINDS and RUDDER) scheme.

A major reason for this is that when a distributed set of Service Agents are monitoring each node, it would be faster to detect failure of service providers than using a single centralized service agent, which has to slice its time in order to monitor each of the services taking part in the composition.

Table 6.9: Simulation Data for No of Tasks that met deadline when Service Provider(s) fail

# Tasks	# Meet Deadline (Centralized)	# Meet Deadline (Decentralized)
2	2	2
4	3	3
6	5	5
8	6	6
10	7	7
12	8	10
14	9	12
16	8	13
18	8	13
20	9	13
22	9	16
24	6	14
26	8	16
28	6	19
30	6	17
32	9	25
34	12	28
36	8	27
38	6	29
40	7	22

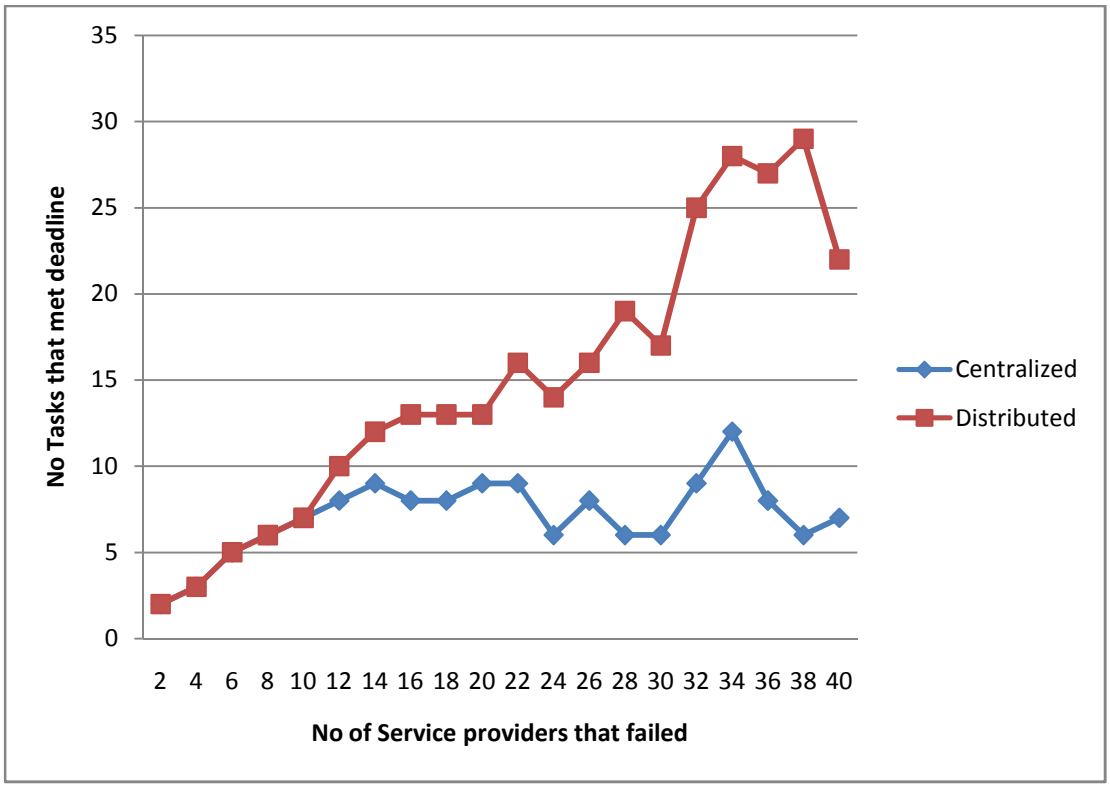

Figure 6.9: No of Tasks that met deadline as no of service provider failure increases

6.4.2.5 Bandwidth Cost as the Number of Services increase

The aim of this experiment was to investigate the effect on the bandwidth cost of an increase in the number of participating services in a composite service. As discussed earlier in sections 6.3, there are two schemes: One, centralized execution mechanism used in conventional workflow such as ACE and RUDDER and decentralized execution found in both choreography and collaboration as found in our own scheme MINDS. The experiments were conducted by varying the number of participating services and programmatically generating the dependency matrix and weighted matrix for the two schemes randomly.

The result of the simulation experiment shows a superior performance in terms of cost-effectiveness in decentralized schemes as against when it is centralized. Figure 6.10 shows the result of the simulation while Table 6.10 presents the simulation data. It was generally observed that the data transmission cost for centralized scheme almost doubles the one for decentralized in the entire range of experimented values. For example, when number of participating services are 41, the overhead cost for data transmissions in both decentralized and centralized scheme are respectively, 136 Rands and 267 Rands. The reason for these is because in a decentralized scheme, data transfers do not pass through a central node during transmission; rather they are delivered directly to the receiving node, thereby reducing the bandwidth overhead due to extra data transmission on the network. The result further confirms that decentralizing the execution as canvassed in MINDS will lead to a more cost-effective service composition strategy.

Table 6.10: Simulation Data for Bandwidth Cost as number of services increase for both Centralized and Decentralized Schemes

No_of_Services	**Total Cost Decentralized (in Rand)**	**Total Cost Centralized (in Rand)**
5	46	81
8	70	122
11	83	151
14	87	155
17	98	176
20	101	190
23	95	182
26	109	202
29	116	221
32	128	244
35	119	229
38	143	255
41	136	247
44	139	251
47	148	280
50	155	292
53	159	301
56	165	312
59	174	330
62	186	356

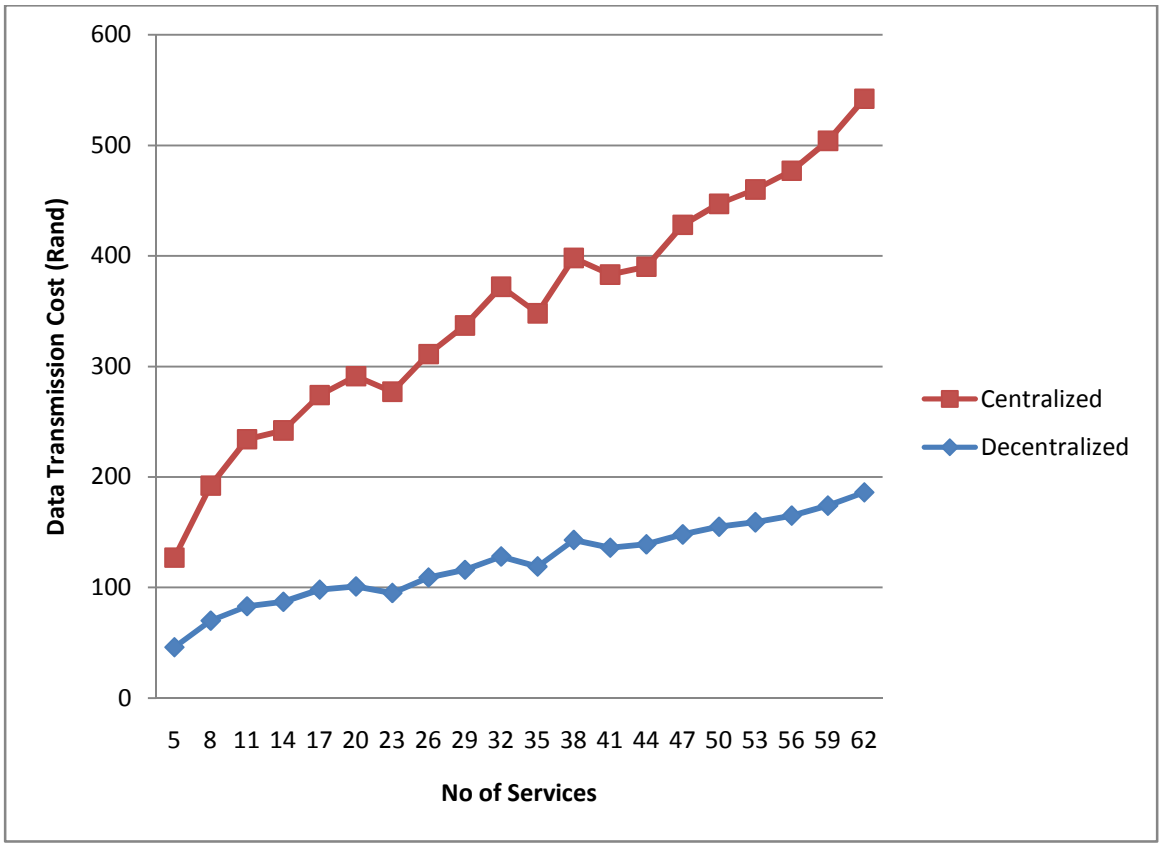

Figure 6.10: Bandwidth Costs for Data Transmission in both Centralized and Decentralized Schemes as no of Services increase

6.5 Chapter Summary

In this chapter, we have evaluated the performance of MINDS by comparing its design decision with some related work in the literature. Specifically, we have compared MINDS with ACE and RUDDER architectures.

We have presented a mathematical analysis of each system's design and show through simulation experiments the performance of MINDS as compared to these other systems.

From the results of the simulation, we observed that MINDS architecture shows performance gain in terms of matchmaking time as the number of services increase thereby making it more scalable. Moreover, MINDS architecture, which is based on Decentralized Matchmaking and Decentralized Execution engine, results in lowest time for composition as the number of task in the composition increase when compared to other systems that are based on Centralized Matchmaking and Centralized Execution engine and Centralized Matchmaking and Distributed Execution engine. Similarly, in terms of fault tolerance, we observed that more tasks meet deadline when either the Service Agent or the Service Provider fails in MIND than both ACE and RUDDER. Finally, MINDS also show superior performance in terms of network bandwidth optimization.

It could be concluded that the use of a centralized broker for service discovery as currently found in many systems leads to performance problem in terms of scalability. The same applies to centralizing the execution agent which is mostly used in existing workflow systems. Decentralization of the matchmaker and the execution again would produce scalable, fault tolerant and a system that is adaptable to low bandwidth as demonstrated in the MINDS architecture.

Chapter 7

Conclusions and Future Directions

This chapter gives a summary of the study, highlighting key achievements and gives directions for further work. The limitations of the study is enumerated which then provide a ground for suggesting areas of our work that we believe requires improvement in the future.

7.1 Summary

The Service Oriented Computing (SOC) paradigm has emerged as an effective approach to engineering distributed applications through service composition. Software applications will rely more and more on reusing existing components on the network, so that composition of applications from various distributed services will be one of the important factors driving future "Internet of Services" (Woods et al, 2009).

Enactment of service compositions demand direct interaction with middleware, which provides support infrastructure and runtime execution environment for services (Sun and Blateky, 2004; Tanenbaum and Van Steen, 2007).

The main goal of this research was to *investigate an effective technique for service composition in dynamic and open environment.*

Currently, most service composition systems are based on workflow technique with more than 100 workflow management systems in use today, but the most popular ones are seen in many industrial and scientific workflow projects such as Triana (Triana, 2003), Taverna (2004), myGrid (2004), and JIGSA (2005).

Using the current state-of-the-art service composition middleware platform where service composition is treated as a workflow has some drawbacks: First, is *lack of flexibility* since in a workflow, service invocation order are predetermined at design time, which means workflows are only constrained to the services whose specifications are available in the platform during design of the workflow. Second, the *lack of scalability* is a result of most workflows relying on a central orchestration engine, but as the number of services involved in the composition increases, using a central coordinator will lead to performance degradation (Bhatia, 2005). Third, workflow-driven composition *lacks adaptation* to user and system requirement changes – yet user and system requirements are rarely static. A workflow's design context might not remain applicable in every detail over the workflow's lifetime. Fourth, interactions among services are limited or poor because services are inherently not communicative. Services are by nature "passive" until they are invoked and they cannot react intelligently to changes in their execution environment.

The foregoing limitations and challenges with current state-of-the-art service composition system were addressed in the approach (called MINDS) presented in this thesis for flexible and adaptive services provisioning with suitability for open and dynamic settings (Iyilade et al, 2009). The following major research contributions have also been reported:

(1) *flexible and adaptive service composition architecture:* Unlike the current state-of-the-art systems, MINDS architecture was centered on dynamically formed team of software agents with collective goals. By complementing current service implementation technologies (grid/web services) with techniques from Agent Oriented Software Engineering (AOSE), services becomes "active" and "alive" to opportunities around and becomes autonomous in taking decision on the tasks to be involved in based on their own volition. We have evaluated the performance of MINDS with respect to strategies of some other related service composition systems such as Rudder (Li and Parashar, 2006) and ACE (Agarwal et al, 2003). A

performance analysis was formulated using *Scalability*, *Fault Tolerance*, and *Network Bandwidth Cost Optimization* as metrics. Various simulations experiments were carried out in Java that investigated the following key performance questions:

a. what is the effect of increasing number of registered services on service discovery time?
b. how does increasing the number of tasks in the composition affect service composition time?
c. how does the failure of service provider(s) affect(s) the number of composition tasks that met deadline?
d. how does the failure of the service monitoring agent(s) impact(s) the number of composition tasks that met deadline?
e. what is the effect of increasing the number of interacting service providers on the bandwidth cost?

The results of the simulation experiments showed that MINDS is scalable because it requires lower service matchmaking and composition time as the number of services and tasks increased respectively. Moreover, in a sample of 80 tasks, 34 (representing 42.5%) met deadline when the centralized service agent used in ACE and Rudder failed, while 55 (representing 68.75%) met deadline when decentralized service agents used in MINDS failed. Also, more tasks met deadline in MINDS when the service provider failed than the centralized service agent approach because the failure was discovered at a minimal time. Finally, the simulation results also indicated that MINDS optimize bandwidth by requiring less amount of data traffic on the network.

(2) *a service provisioning life-cycle based on composition goal and service preconditions:* We also presented a life-cycle process for service provisioning based on MINDS. The service provisioning life-cycle process was based on just preconditions and goals to be achieved by the composition. Unlike existing

systems, Services were not bound to the application at design time. The entry to the life-cycle was a user request and output was the result of composition to the user. The life-cycle comprised a four-step process - Task Decomposition and Representation; Dynamic Service Matchmaking; Composition Synthesis; Execution and Monitoring. Besides, the infrastructure was able to react to any partial failure in the system during execution. To test the applicability of the life-cycle process, an experimental prototype was implemented and demonstrated for the Planit (Pty) Ltd itinerary planner illustrating a use-case for MINDS in real-life settings especially for dynamically formed Virtual Organization ecosystem where business services are shared in an ecosystem.

7.2 Limitations of the Study

The middleware infrastructure presented in the thesis is only an experimental testbed for us to elaborate our service composition strategy and it is not yet ready for immediate use in a production environment. There are a number of limitations of the work in its current form, which presents us good opportunity for further work in the future. These limitations are enumerated below:

First, we have only used a very simplistic negotiation strategy based on Contract Net Protocol in our model for the choice of service providers, many real-life systems present more serious challenges that would require advanced negotiation and market mechanisms both in time and space for making such decisions. We do not consider such cases in our model.

Two, when resources (services) are shared in open environment, trust and security are vital issues. How do we ensure that malicious entities are not going to take advantage of the system for dubious means? How do we classify users of the system and define access privileges? While, these questions are vital in real-life deployment of MINDS, our work was simply an experimental prototype with specific focus on addressing the service composition problem and did not address these issues.

Three, another salient issue which we did not address is the Service Level Agreement between service provider and service consumer. How do we ensure that service providers will deliver on their promise and guarantee Quality of Service (QoS)? Without this, it becomes difficult to guarantee that service providers will honour their contractual obligations and meet consumer's expectation on a service.

Four, we also did not address the economic aspect of service usage. That is, how are service usages going to be metered, charged and paid for? Is it per transaction, flat-fee, or subscription-based? And which protocol will be employed to facilitate this?

Five, we have also use a simplistic experimental setup for our prototype implementation with only a handful services as discussed in Chapter 5. Our experiments were carried out using localhost resources, and we do not deploy our system on a global service oriented infrastructure which may give us a more realistic analysis of our system.

7.3 Future Work

In view of the limitations identified in Sections 7.2 above, a number of improvements and extensions are worth pursuing in our future work.

Obviously, it would be interesting to investigate and incorporate some other advanced *negotiation* strategies in the Service Provisioning Life Cycle presented in Chapter 4. Such negotiation allows for implementation of market mechanisms for proposals, trading and offering concession. It can also define a wide range of issues over which participants must agree such as cost, response time and penalties in case of breach of commitment. Interestingly, Software agents provide many useful metaphors for automating support of intelligent negotiations. Agents facilitate dynamic choice of negotiation commitments and enables adjustment of negotiation behaviours at runtime (Singh and Huhns, 2005).

Furthermore, the issues of *trust* and *security* in MINDS are also worth pursuing in order to facilitate the deployment of MINDS in a real-life setting.

SOC environments pose greater challenge to security because it inherently involves interactions among autonomous entities (Huhns and Singh, 2005). It would be interesting to examine how current web services standards for security such as WS-Security, WS-Trust (W3C, 2002a) could be incorporated into how services are accessed.

Another interesting extension of this work is the incorporation of a *Service Level Agreement (SLA)* framework in the Service Provisioning Life Cycle, especially before binding to services. Service Level Agreement is an instrument that defines the relationship between the providers and consumers. This serves to guarantee service quality and manage consumer expectations.

In addition, it is also important that we address the issue of how services are going to be charged and paid for. Although, we have adopted the *utility model* in this research which means service consumers are going to be charged per use rather than a flat rate mechanism. Currently, there are economic and non-economic approach to encourage usage and provisioning of services. Our previous works in Iyilade, et al (2007) and Buthelezi et al (2008) provides us a good starting point to address the issue of accounting and charging for service usage.

Finally, we intend to extend some of our models in the future by applying them to more realistic usage scenarios and deploying them on a real life SOC test-bed. Our next interest in this respect is to incorporate MINDS into our center's Niche Research projects, Grid-based Utility Infrastructure for SMME-enabling Technologies (GUISET) (Adigun et al, 2006), and the SMME E-commerce on-Demand (SECOND).

Appendices

A. Selected Publications

- Adigun, M.O. **Iyilade J.S.**, Kabini, K. (2010). Agent-based Infrastructure for Dynamic Composition of Grid Services. A chapter contribution to *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications*. N. Antonopoulos, G. Exarchakos, M. Li, A. Liotta (Eds). Pp. 911-936. IGI-Global Publishing.

- **Iyilade, J.S.,** Kabini, K., Adigun, M.O (2009). MINDS: A middleware Infrastructure for Distributed Services Provisioning. *Proceedings, IEEE 6th International Conference on Information Technology: New Generations, April 27-29, 2009, Las Vegas, Nevada, USA. pp* 1018 – 1023.

- Buthelezi, M.E., **Iyilade J.S**., Adigun, M.O. (2008) "Pricing and Charging Models for Next Generation e-Services", In Proceeding of 10th WWW Conference, Cape town, Sept 3-5, 2008.

- **Iyilade J.S.**, Migiro S.O., Aderounmu G.A., Adigun M.O (2007). Using Grid Computing for Collaborative e-Commerce among SMMEs. *Proceedings of 9th World Wide Web conference, Johannesburg, 5-7 Sept. 2007*. ISBN: 978-0-620-39837-4

- Kabanda, S.K. **Iyilade, J.S**. Adigun, M.O (2007). Knowledge Resource Providers in a Grid-enabled Infrastructure – The case of the deep rural SMMEs in South Africa. *In Proceedings of the 4th International Conference on Intellectual Capital and*

Knowledge Management (ICICKM, 2007), 15 - 16 Oct 2007, Cape Town, South Africa, pp 199-208

- **Iyilade, J.S.** Aderounmu, G.A. Adigun, M.O (2007). Incentives for Resource Sharing and Cooperation in Grid Computing System. *Proceedings of the IEEE International conference on Next Generation Mobile Applications, Services and Technologies (NGMAST '07), September 11-14, Cardiff, Wales, UK,* pp 191-196.

B: Sample Source Code for the Simulation Experiments

(a) Fault tolerance

```
//import java.awt.*;

import java.util.*;
import java.io.*;
/**
 *
 * <P>
 * @author Seun Iyilade
 */

public class FaultTolerance {  //1

    // public static int REGISTEREDSERVICES =  10; // No of Services in the Registry pool

     public static int NOOFTASKS = 10; // Total no of tasks in a composition request instance

     Random rand;

   CentralizedCoordinator centralized;
   DistributedCoordinator distributed;

 public FaultTolerance(){

    centralized  = new CentralizedCoordinator();
    distributed = new DistributedCoordinator();



    runCode();
 }

 public void runCode(){

 String noTasks;  // No of Services ...
  String taskCentralized;
  String taskDistributed;

  int dataCentralized ;
  int dataDistributed ;

  int[] t_exec; //array of task execution time on a particular node
  int[] t_spent; // array of time already spent in execution before error
  int[] lftime;
```

```
  int[] mttr;

    //int noofServices ;
  int noofTasks;

  System.out.println("NOOFTASKS" + "\t" + "CENTRALIZED" + "\t" + "DISTRIBUTED"
+"\n");

System.out.println("=====================================
====================" + "\n");

 try
   {  //3

   FileOutputStream file = new FileOutputStream ("faulttolerance.txt", true);
   String headings = "NOOFTASKS" + "\t" + "CENTRALIZED" + "\t" + "DISTRIBUTED"
+"\n";
   file.write (headings.getBytes());

    //noofServices = REGISTEREDSERVICES;
    noofTasks = NOOFTASKS;

    rand = new Random();

   for (int j = 0; j<20 ; j++)

     { //4

       t_exec = new int[noofTasks];
       t_spent = new int [noofTasks];
       lftime = new int [noofTasks];
       mttr = new int [noofTasks];

       //filling the arrays with default data values

       for(int k=0; k< t_exec.length; k++){

          t_exec[k]= randomGen(300);   //execution time for task [1...300secs]

          t_spent[k] = randomGen(t_exec[k]);  // time spent before error  [1...180secs]



          lftime[k] = (int) (t_exec[k] + Math.abs(0.3 * t_exec[k]));

          mttr[k] = randomGen(120);


       }

   dataDistributed =  distributed.resultGenerator(noofTasks,t_spent,lftime,mttr);
   dataCentralized = centralized.resultGenerator(noofTasks,t_spent,lftime);

   noTasks = String.valueOf (noofTasks);
   taskDistributed = String.valueOf(dataDistributed);
```

```
    taskCentralized = String.valueOf(dataCentralized);
    System.out.println(noofTasks + "\t" + dataCentralized + "\t" + dataDistributed + "\t\n");
    String output = noTasks + "\t" + taskCentralized + "\t" + taskDistributed + "\n";
    file.write(output.getBytes());
    noofTasks += 5;
      } //4
     file.close();
       } //3
    catch (IOException ex){ex.printStackTrace();}
     //2


}
public int randomGen (int seed){
return 1 + rand.nextInt(seed);
}
 public static void main(String[] args) { //2

   new FaultTolerance();

 }
 } //1

class DistributedCoordinator{

 public DistributedCoordinator(){
    }
public int resultGenerator(int noofTasks, int[] t_spent,int[] lftime,int[] mttr){
    //this.noServices = noServices;
   int noMeetDeadline = 0;
   for (int i=0; i<noofTasks;i++){
   int val = t_spent[i]+mttr[i];
```

```
    if (val < lftime[i])
       noMeetDeadline +=1;
    }

     //now sort and find the highest lateST, that is the max time to discover all services

    return noMeetDeadline;

     }

    }


class CentralizedCoordinator {

    int mttr = 120;

 public CentralizedCoordinator(){


      }

public int resultGenerator(int noofTasks, int[] t_spent,int[] lftime){

    int noMeetDeadline = 0;

    for (int i=0; i<noofTasks;i++){

    int val = t_spent[i]+mttr;

    if (val < lftime[i])
       noMeetDeadline +=1;
    }



     return noMeetDeadline;

}

}
```

(b) Service Composition Time

```
    import java.util.*;
    import java.io.*;

    /**
     * @author Seun
     */

    public class ServiceCompositionTime {
```

```
   static int noofTasks;
   static int noofServices;

   int cmValue;
   int dmValue;
   int ceValue;
   int deValue;

   CentMatch cm; // Case 1: Centralized Matchmaking
   DistMatch dm; // Case 2: Distributed Matchmaking
   CentExec ce;  // Case 3: Centralized Execution Engine
   DistExec de;  //Case 4: Distributed Execution Engine

 public ServiceCompositionTime(){

      cm  = new CentMatch();
      dm = new DistMatch();
      ce = new CentExec();
      de = new DistExec();

      getResult();

    } //2


  public void getResult(){

   String noTasks;  // No of Services ...
   String timeCMCE;
   String timeCMDE;
   String timeDMDE;

   int dataCMCE;
   int dataCMDE;
   int dataDMDE;

   System.out.println("NOOFTASKS" + "\t" + "CMCE" + "\t" + "CMDE" + "\t" + "DMDE"
+"\n");

System.out.println("=================================================
==================" + "\n");

  try
    {  //3

   FileOutputStream file = new FileOutputStream ("servcomptime.txt", true);
   String headings = "NOOFTASKS" + "\t" + "CMCE" + "\t" + "CMDE" + "\t" + "DMDE"
+ "\n";
   file.write (headings.getBytes());

    noofServices = 100;
    noofTasks = 1;


 for (int j = 0; j<20 ; j++)
```

```
{ //4
   setCMTime();
   setDMTime();
   setCETime();
   setDETime();


dataCMCE = cmValue + ceValue;
dataCMDE = cmValue + deValue;
dataDMDE = dmValue + deValue;


noTasks = String.valueOf (noofTasks);
//noTasks = String.valueOf(noofServices);
timeCMCE = String.valueOf(dataCMCE);
timeCMDE = String.valueOf(dataCMDE);
timeDMDE = String.valueOf(dataDMDE);

System.out.println(noofTasks + "\t" + dataCMCE + "\t" + dataCMDE + "\t" +
dataDMDE + "\t\n");

String output = noofTasks + "\t" + timeCMCE + "\t" + timeCMDE + "\t" +
timeDMDE + "\n";

file.write(output.getBytes());

noofTasks += 1;
//noof += 5;
  } //4

 file.close();

  } //3

catch (IOException ex){ex.printStackTrace();}

}


public void setCMTime(){

   cmValue = cm.resultGenerator(noofServices, noofTasks);

}

public int getCMTime(){

   return cmValue;

}

public void setDMTime(){

   dmValue = dm.resultGenerator(noofServices, noofTasks);
```

```
    }

    public int getDMTime(){

       return dmValue;

    }

    public void setCETime(){

       ceValue = ce.resultGenerator(noofTasks);

    }

    public int getCETime(){

       return ceValue;

    }

    public void setDETime(){

       deValue = de.resultGenerator(noofTasks);

    }

    public int getDETime(){

       return deValue;

    }


public static void main(String[] args) { //2

    new ServiceCompositionTime();

     } //1
}

class CentMatch{

    int noServices;
    Random rand3;

public CentMatch(){
       rand3 = new Random();
   }

public int resultGenerator(int noServices,int noofTasks){

    int totalTime = 0;
    double searchTime = 10.0; //searchtime per service in seconds
```

```
    this.noServices = noServices;

  for (int k=0;k<noofTasks;k++){

     int p = randomgenerator();

     totalTime += Math.ceil(p*searchTime);  // totalTime = Sum (p*St) for all tasks

  }
    return totalTime;

}


public int randomgenerator(){

   int val = 1 + rand3.nextInt(noServices);
  // System.out.println ("val =" + val);

   return val;

  }
 }


class DistMatch{

   int lateST[]; //declare an int array to store latest search time for each GSA for task in
tuple space
   int noServices;
   Random rand3;

 public DistMatch(){

    rand3 = new Random();
     }

public int resultGenerator(int noServices,int noofTasks){

   this.noServices = noServices;

    int maxVal = 240; //set maximum check time for service as 3mins (=180secs);
    lateST = new int[noServices]; //create array for equal to no of services


    //this array generate random values for each GSA

    for (int m=0;m<noServices;m++){

    lateST[m] = randomgenerator(maxVal);

    }

    //now sort and find the highest lateST, that is the max time to discover all services
```

```
    sort();

    //tell me the highest value
  //  System.out.println ("The Highest =  "+ lateST[0]);

    return lateST[0];

}

public void sort(){

   int biggest;

   for (int i=0; i<lateST.length -1; i++){
   biggest = i;

      for (int index = i+1; index < lateST.length; index++)
         if (lateST[index] > lateST [biggest])
            biggest = index;

    swap (i,biggest);
     }
   }

public void swap (int first, int second){

 int temporary = lateST [first];
 lateST[first] = lateST [second];
 lateST[second] = temporary;

}


public int randomgenerator(int myVal){

   int val = 1 + rand3.nextInt(myVal);
  // System.out.println ("val =" + val);

   return val;

  }
}


class CentExec {

    Random rand2;
    static int execTime;
    int noofTasks;

 public CentExec(){

     rand2 = new Random();
     execTime = 180; //set maximum execution time to 60 seconds
      }
```

```
public int resultGenerator(int noofTasks){

   this.noofTasks = noofTasks;
   int totalTime = 0;

   for (int k=0;k<noofTasks;k++){

     int t = randomgenerator();

     totalTime += t;  // totalTime = Sum (p*St) for all tasks

  }
    return totalTime;

}
public int randomgenerator(){

  int val = 1 + rand2.nextInt(execTime);
 // System.out.println ("val =" + val);

  return val;
 }
}

class DistExec {

   Random rand2;
   int execTime;
   int noofTasks;
  int taskNewExecTime[];

 public DistExec(){

    rand2 = new Random();
    execTime = 180;
     }

public int resultGenerator(int noofTasks){


   this.noofTasks = noofTasks;
   int taskExecTime[] = new int[noofTasks]; //declare an array to store task execution
time
   int taskType[]=  new int [noofTasks]; //an array to store task type i.e dependent =1,
independent =0

    taskNewExecTime = new int[noofTasks]; // for modified task exec time calculation


   //generate value for execution time and task type

   for (int k=0;k<noofTasks;k++){

     taskExecTime[k] = randomgenerator(execTime);
```

```
    taskType[k] = randomgenerator();

     }

   //now recalculate the executiontime based on task type
   taskNewExecTime[0] = taskExecTime[0];

   for (int m=1;m<noofTasks;m++){

     //test if task is a dependent task and not index must be greater than 0
     if (taskType[m]== 1){
   taskNewExecTime[m] = taskExecTime[m] + taskExecTime[m-1];
    //  System.out.println("this is task:  " + m + "  and has dependency");
    // System.out.println("Tasktype: " + taskType[m]);
     }

     else {
        taskNewExecTime[m] = taskExecTime[m];
     //   System.out.println("this is task: " + m + "  and has no dependency");
      //  System.out.println("Tasktype: " + taskType[m]);
     }

     }

   //now do the sorting for taskNewExecTime based on highest exec time

 sort();

 return taskNewExecTime[0]; //returns the first taskExecTime which is the highest
after sorting

}

public void sort(){


  int biggest;

  for (int i=0; i<taskNewExecTime.length -1; i++){
  biggest = i;

     for (int index = i+1; index < taskNewExecTime.length; index++)
        if (taskNewExecTime[index] > taskNewExecTime[biggest])
           biggest = index;

   swap (i,biggest);
    }
  }

public void swap (int first, int second){

 int temporary = taskNewExecTime [first];
 taskNewExecTime[first] = taskNewExecTime [second];
 taskNewExecTime[second] = temporary;
```

```
    }


    public int randomgenerator(){

      int val =  rand2.nextInt(2);

     return val;

     }

    public int randomgenerator(int execTime){

      int val = 1 + rand2.nextInt(execTime);
     // System.out.println ("val =" + val);

      return val;
     }
    }
```

(c) Scalability

```
//import java.awt.*;

import java.util.*;
import java.io.*;
/**
 *
 * <P>
 * @author Seun Iyilade
 */

public class Scalability {  //1

    public static int REGISTEREDSERVICES =  10; // No of Services in the Registry pool

    public static int NOOFTASKS = 5; // Total no of tasks in a composition request
instance

    Random rand;

 public Scalability(){
        rand = new Random();
 }

  public static void main(String[] args) { //2


   CentralizedMatchmaker centralized  = new CentralizedMatchmaker();
   DistributedMatchmaker distributed = new DistributedMatchmaker();

   new Scalability();
```

```
  String noServices;  // No of Services ...
  String timeCentralized;
  String timeDistributed;

  int dataCentralized ;
  int dataDistributed ;

  int noofServices ;
  int noofTasks;

  System.out.println("NOOFSERVICES" + "\t" + "CENTRALIZED" + "\t" + "DISTRIBUTED"
+"\n");

System.out.println("=====================================
====================" + "\n");

 try
   {  //3

   FileOutputStream file = new FileOutputStream ("scalability.txt", true);
   String headings = "NOOFSERVICES" + "\t" + "CENTRALIZED" + "\t" + "DISTRIBUTED"
+"\n";
   file.write (headings.getBytes());

    noofServices = REGISTEREDSERVICES;
    noofTasks = NOOFTASKS;

   for (int j = 0; j<20 ; j++)

     { //4

   dataDistributed =  distributed.resultGenerator(noofServices,noofTasks);
   dataCentralized = centralized.resultGenerator(noofServices,noofTasks);

   noServices = String.valueOf (noofServices);
   timeDistributed = String.valueOf(dataDistributed);
   timeCentralized = String.valueOf(dataCentralized);

   System.out.println(noofServices + "\t" + dataCentralized + "\t" + dataDistributed +
"\t\n");

   String output = noServices + "\t" + timeCentralized + "\t" + timeDistributed + "\n";

   file.write(output.getBytes());

   noofServices += 5;

     } //4

    file.close();

     } //3

   catch (IOException ex){ex.printStackTrace();}
```

```
    } //2


  } //1


class DistributedMatchmaker{

    int lateST[]; //declare an int array to store latest search time for each GSA for task in
tuple space
   int noServices;
   Random rand3;

 public DistributedMatchmaker(){

     }

public int resultGenerator(int noServices, int noofTasks){

    this.noServices = noServices;

    int maxVal = 240; //set maximum check time for service as 3mins (=180secs);
    lateST = new int[noServices]; //create array for equal to no of services


    //this array generate random values for each GSA

    for (int m=0;m<noServices;m++){

    lateST[m] = randomgenerator(maxVal);

    }

    //now sort and find the highest lateST, that is the max time to discover all services

   sort();

   //tell me the highest value
 //  System.out.println ("The Highest =  "+ lateST[0]);

   return lateST[0];

    }

public void sort(){

  int biggest;

  for (int i=0; i<lateST.length -1; i++){
  biggest = i;

     for (int index = i+1; index < lateST.length; index++)
        if (lateST[index] > lateST [biggest])
           biggest = index;
```

```
    swap (i,biggest);
     }
   }

public void swap (int first, int second){

 int temporary = lateST [first];
 lateST[first] = lateST [second];
 lateST[second] = temporary;

}

public int randomgenerator(int maxVal){
   rand3 = new Random();

   int val = 1 + rand3.nextInt(maxVal);
   //System.out.println ("val =" + val);

   return val;

  }

    }


class CentralizedMatchmaker {

    Random rand2;
    int noServices;

 public CentralizedMatchmaker(){

     rand2 = new Random();
      }

public int resultGenerator(int noServices,int noofTasks){

    int totalTime = 0;
    double searchTime = 10.0; //searchtime per service in 30 seconds

     this.noServices = noServices;

  for (int k=0;k<noofTasks;k++){

      int p = randomgenerator();

      totalTime += Math.ceil(p*searchTime);  // totalTime = Sum (p*St) for all tasks

  }
     return totalTime;

}
```

```
public int randomgenerator(){

   int val = 1 + rand2.nextInt(noServices);
  // System.out.println ("val =" + val);

   return val;

  }
}
```

Bibliography

Adigun, M.O. Emuoyibofarhe, O.J. Migiro, S.O. (2006). Challenges to Access and Opportunity to use SMME enabling Technologies in Africa. *1st All-African Technology Diffusion Conference*, Johannesburg, South Africa. June 12-14.

Agarwal, M. Bhat, V. Li, Z. Liu, H. Matossian, V. Putty, V. Schmidt, C. Zhang, G. Parashar, M. Khargharia, B. and Hariri, S. AutoMate: Enabling Autonomic Applications on the Grid. In Proc of Autonomic Computing Workshop, 5th Annual International Active Middleware Services Workshop(AMS 2003), pages 365-375, Seattle, WA, June 25 2003.

Akkiraju, R. Farrell, J. Miller, J. Nagarajan, M. Schmidt M. Sheth, A. & Verma, K. (2006). Web service semantics: WSDLS. Retrieved February 20, 2007 from http://www.w3.org/Submission/WSDL-S.

Alamri, A. Eid, M. and El Saddik, A. (2006). Classification of state-of-the-art dynamic web services composition techniques. *International Journal of Web and Grid Services,* Vol. 2, No 2, 2006.

Andreas, P & Peter, T (2008). Service Infrastructure. In: Semantic Service Provisioning. Dominik K.; Peter T.; Steffen S, Mathias W (Eds.). Springer Publisher. ISBN 978-3-540-78616-0

Apache, (1998). Online at: http://tomcat.apache.org/. Last Accessed: 15th April, 2010

Apache, (2003) http://ws.apache.org/juddi/. Last Accessed: 15th April, 2010

Barker A, Weissman J.B and van Hemert, J. (2009b). Avoiding Workflow Bottlenecks Caused By Centralised Orchestration. *Springer Journal of Cluster Computing*. the response can be generated as a J2ME[18] file.
Vol 12. No 2. Pp 221-235.

Barker, A. Walton, C.D. and Robertson, D. (2009a). Choreographing Web Services. *IEEE Transactions on Service Computing.* Vol 2 No. 2.

Beiriger, J.I. Johnson, W.R. Bivens, H.P. Humphreys, S.L. & Rhea, R. (2000). Constructing the ASCI Computational Grid. *Proceedings of the Ninth International Symposium on High Performance Distributed Computing (HPDC 2000)*, Pittsburgh, CS Press.

Berardi, D. Calvanese, D. De Giacomo, G. Mecella, M. (2005). Composition of services with nondeterministic observable behaviour. In Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC '05), Lecture Notes in Computer Science, vol 3826, pages 520-526. Springer, New York.

Berners-Lee, T. Hendler, J. & Lassila, O (2001). The Semantic Web. *Scientific American*, 34-43.

Bhatia, K. (2005). "Peer-To-Peer Requirements On The Open Grid Services Architecture Framework". GGF document by the OGSAP2P Research Group. Accessed online at: www.ggf.org

BPML (2002). Online at: www.ebpml.org/bpml.htm. Last accessed: 10th November, 2008

Bradshaw, J. M. (1997). An introduction to software agents. In J. M. Bradshaw (Ed.), *Software agents* (pp. 3-46). AAAI Press/The MIT Press.

Brebner, P. Emmerich, W. Two ways to Grid: The contribution of Open Grid Services Architecture (OGSA) mechanisms to service-centric and resource-centric lifecycles. *Journal of Grid Computing* 2006; 4(1):115–131.

Burbeck, S (2000). The tao of e-business services – the evolution of web applications into service-oriented components with web services. Technical report, IBM Software Group, October 2000. Online at: http://www.ibm.com/developerworks/webservices/library/ws-tao/

Buthelezi, M.E. Iyilade, J.S. Adigun, M.O. (2008) "Pricing and Charging Models for Next Generation e-Services", *In Proceeding of 10th WWW Conference*, Cape town, Sept 3-5, 2008.

Cao, J. (2001). Agent-based Resource Management for Grid Computing. Unpublished doctoral dissertation. University of Warwick, Coventry, UK

Cardoso, J. and Sheth A. (2003). Semantic e-Workflow Composition. *Journal of Intelligent Information Systems (JUS).* ZIO**:** 191-225.

Cardoso J. and Sheth A.P. (2006). The Semantic Web and its Applications. In: Semantic Web Services, Processes and Applications. Cardoso J. and Sheth A.P. (Eds.). Springer.

Castelfranchi, C., Miceli, M., & Cesta, A. (1992). Dependence relations among autonomous agents. Proceedings of the 3rd European Workshop on Modeling Autonomous Agents and Multi-Agent Worlds (pp. 215-231). Amsterdam, The Netherlands: Elsevier Science Publisher.

Curbera, F. Goland, Y. Klein, J. Leymann, F. Roller, D. Thatte, S. Weerawarana, S (2002). Business Process Execution Language for Web Services, Version 1.0 Online at: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel1.pdf. Last accessed: 10th November, 2008

Cybok, D. (2004). A Grid Workflow Infrastructure, Presentation in GGF-10, Berlin: Germany.

DAML (2001). The DARPA Agent Markup Language. Online at: http://www.daml.org

De Roure, D. Jennings, N.R. & Shadbolt, N.R. (2001). The Semantic Grid: A future e-Science infrastructure. Available online at: http://www.semanticgrid.org/documents/semgrid-journal/semgrid-journal.pdf. Last accessed: 15th May, 2007

EC-ISU (2006). Software, Services and Complexity Research in IST Programme. Software Technologies Unit, Information Society and Media DG, European Commission. Accessed online at: http://cordis.europa.eu/st/index.html. Last accessed date: November 10th 2008.

Endrei, M. Ang, J. Ansanjani A. Chua, S. Comte, P. Krogdahi, P. Luo, M. & Newling, T. (2004). Patterns: Service-Oriented Architecture and Web Services" IBM Redbooks. Online at: www.ibm.com/redbooks. Last accessed: 17th May, 2007

Fluegge M.; Garcia dos Santos I.J.; Tizzo N.P.; Madeira E.R. (2006). Challenges and Techniques on the Road to Dynamically Compose Web Services. In proceeding ICWE'06, ACM 1-59593-352-2/06/0007

Foster, I. (2005). "Globus Toolkit 4: Software for Service Oriented Systems". In H. Jin, D. Reed, and W. Jiang (Eds.): NPC 2005, LNCS 3779, pp. 2 – 13. Accessed online at: http://www.globus.org/alliance/publications/papers/IFIP-2005.pdf. Last accessed: 10th November, 2008.

Foster, I. Jennings, N.R. and Kesselman, C. (2004). Brain Meets Brawn: Why Grid and Agents Need Each Other. Proceedings of Third International Conference on Autonomous agents and Multi-agent Systems. Pp 8-15

Foster, I. and Kesselman, C. (1998). The Grid: Blueprint for a New Computing Infrastructure. San Francisco, CA: Morgan Kaufmann Publishers.

Foster, I. Kesselman, C. and Tuecke, S. (2001). The Anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3), 200-222, www.globus.org/research/papers/anatomy.pdf.

Freeman, E. Hupfer, S. and Arnold, K. (2008). JavaSpaces Principles, Patterns, and Practice. Addison-Wesley.

Gelenter, D. (1985). Generative Communication in Linda. *ACM Transaction of Prog. Lang. Syst.* (7)1: 80-112.

Genesereth, M. R., & Ketchpel, S. P (1994). Software agents. *Communications of the ACM,* 37(7), 48-53.

GGF (2002). Online at: http://www.gridforum.org

Globus Alliance (2005). "Globus Toolkit 4 Java WS Core" Online at: http://www.globus.org/toolkit/docs/4.0/common/javawscore/

Grossman P. (2002). Discrete Mathematics for Computing (2nd edition). Pelgrave: Macmillan Press.

Huang, H. & Walker, D. (2003). Extensions to Web Service Techniques for Integrating Jini into a Service-Oriented Architecture for the Grid. Proceedings of the International Conference on Computational Science 2003 (ICCS '03), Melbourne, Australia. Lecture Notes in Computer Science, Springer-Verlag.

Huhns, M.N & Singh, M.P (2005). Service-Oriented Computing: Key Concepts and Principles, *IEEE Internet Computing*, Vol. 9, No. 1, 2005, pp. 75-81.

Huhns, M.N. Singh, M.P. Burstein, M. Decker, K. Durfee, K.E. Finin, T. Gasser, T.L. Goradia, H. Jennings, P.N. Kiran L. Nakashima, H. van Dyke Parunak, H. Rosenschein, J.S. Ruvinsky, A. Sukthankar, G. Swarup, S. Sycara, K. Tambe, M. Wagner, T. Zavafa, L. (2005) Research Directions for Service-oriented Multiagent Systems. *Internet Computing*, IEEE. Volume 9, Issue 6, Nov.-Dec. 2005 Page(s): 65 – 70.

IBM. (n.d). Online at: http://www.ibm.com/developerworks/webservices/standards/. Last accessed: 17th October, 2008.

Issarny, V. Caporuscio, M. Georgantas, N (2007). A Perspective on the Future of Middleware-based Software Engineering. *In proceedings of IEEE conference on Future of Software Engineering* (FOSE 07).

Iyilade, J.S. Kabini, K. Adigun, M.O. (2009). MINDS: A middleware Infrastructure for Distributed Services Provisioning. Proceedings, IEEE 6th International Conference on Information Technology: New Generations, April 27-29, 2009, Las Vegas, Nevada, USA. pp 1018 – 1023.

Iyilade, J.S. Migiro, S.O. Aderounmu, G.A. Adigun M.O. (2007). Using Grid Computing for Collaborative e-Commerce among SMMEs. In Proceedings of 9th World Wide Web conference, Johannesburg, 5-7 Sept. 2007. ISBN: 978-0-620-39837-4

JADE (2005), Online at: http://jade.tilab.com/. Last Accessed: 15th April, 2010

Jennings, N. R. Norman, T. J. Faratin, P. & Odgers, B. (2000). Autonomous agents for business process management. *International Journal of Applied Intelligence*, 14(2), 145-189

Jennings, N.R. & Wooldridge, M.J. (1998). Agent technology: Foundations, Applications and Markets. Springer-Verlag.

JIGSA (2005). URL: http://www.gridworkflow.org/snips/gridworkflow/space/JIGSA. Last accessed date: 10th June, 2009.

JINI (1994). URL: http://www.sun.com/software/jini/. Last Accessed: 15th April, 2010

Juric, M.B. Mathew, B. and Sarang, P. (2006). Business Process Execution Language for Web Services (2nd Eds). Packt Publishing Ltd.

Klein, M. and A. Bernstein (2001). Searching for Services on the Semantic Web Using Process Ontologies. International Semantic Web Working Symposium (SWWS), Stanford University, California, Springer.

Koehler, J. & Alonso, G. (2007). Service Oriented Computing. Ercim News volume 70, pp 15-16, July 2007.

Kowalkiewicz, M. Ludwig, A. Meyer, H. Schaffner, J. Stamber, C. and Stein, S. (2008). Service Composition and Binding. In: Semantic Service Provisioning. Dominik K.; Peter T.; Steffen S, Mathias W (Eds.). Springer Publisher. ISBN 978-3-540-78616-0

Krishnan, S. Wagstrom, P. & Laszewski, G. (2002). GSFL: A Workflow Framework for Grid Services. Preprint ANL/MCS-P980-0802, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, USA.

Lee, J.N. Huynh, M.Q. Kwok, R.C.W. Pi, S.M. (2003). IT outsourcing evolution: past, present and future. *Communications of the ACM* 46(5), 84-89

Li, Z. & Parashar, M. (2006). An Infrastructure for Dynamic Composition of Grid Services. Online at: www.caip.rutgers.edu/TASSL/Papers/rudder-grid-06.pdf. Last Accessed: 15th April, 2010.

Mendoza, A. (2007). Utility Computing – Technologies, Standards, and Strategies. Artech House Publishing. ISBN-13-978-1-59693-024-7

Muller, I. Kowalczyk, R. Braun, P. (2006). Towards Agent-based Coalition Formation for Service Composition. Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology. Online at: ieeexplore.ieee.org/iel5/4052878/4052879/04052901.pdf. Last Accessed: 15th April, 2010.

myGrid (2004). URL: http://www.mygrid.org.uk/. Last accessed date: June 10th, 2009

Nau, D. Au, T. Ilghami, O. Kuter, U. Murdock, J. Wu, D. Yaman, F. (2003). SHOP2: An HTN planning system, Journal of Artificial Intelligence Research 20. 379–404.

Narayanan, S and Mcllraith, S.A (2002). Simulation, verification and automated composition of web services. Proceedings of the 11th international conference on World Wide Web, Pgs 77 - 88. ISBN:1-58113-449-5

Nessi-Grid (2006). Networked European Software and Services Initiative – Grid. Grid vision and strategic research agenda. Accessed online at: www.nessi-europe.com/

NGG Report. (2006). Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Vision and Research Directions 2010 and Beyond. Online at: semanticgrid.org/NGG3.

Nwana, S. Lee, L. Jennings, N.R. (1996). Coordination in Software Agent Systems. *BT Technology Journal*. Vol 14 No 4.

Nwana, H.S. Rosenschein, J.; Sandholm, T.; Sierra, C.; Maes, P. & Guttman, R. (1998). Agent-mediated electronic commerce: Issues, challenges, and some viewpoints. Proceedings of 2nd ACM International Conference on Autonomous Agents, pp 189-196.

Omer, A.M and Schill A. (2009). Web Services Composition using Input/Output Dependency Matrix. In ACM Proceeding AUPC 2009. July 13-17, London. Pp 21-26.

OWL-S. (2004). OWL-based Web service ontology. Retrieved February 20, 2007. From http://www.daml.org/services/owl-s/

Pistore, M. Barbon, F. Bertoli, P. Shaparau, D. Traverso, P (2004). Planning and monitoring Web Service Composition. Lecture Notes in Computer Science, 3192: 106-115.

Polze, A and Troger, P (2008). Service Infrastructure. In: Semantic Service Provisioning. Dominik K.; Peter T.; Steffen S, Mathias W (Eds.). Springer Publisher. ISBN 978-3-540-78616-0.

Ponnekanti, S.R. and Fox. A. (2002). SWORD: A Developer Toolkit for Web Service Composition. Proc. 11th Intl. World Wide Web Conference, Honolulu, Hawaii, May 7-11. Online at: http://radlab.cs.berkeley.edu/people/fox/static/pubs/pdf/c09.pdf

Rodriguez, A. and Egenhofer M. (2002). Determining Semantic Similarity Among Entity Classes from Different Ontologies. IEEE Transactions on Knowledge and Data Engineering. Vol. 15. No. 2. Pp. 442-456

Sheth, A. and Meersman R. (2002). Amicaloia Report: Database and Information Systems Research Challenges and Opportunities in Semantic Web and Enterprises. SIGMOD Record 31(4)**:** pp. 98-106.

SHOP2. (2004). Online: http://www.cs.umd.edu/projects/shop/

Singh, M. P and Huhns, M. (2005). Service-Oriented Computing: Semantics, Processes, Agents. England: John Wiley & Sons.

Singh, M.P. (2004). Agent-based Abstractions for Software Development - Accommodating Complexity in Open Systems in *Methodologies and Software Engineering for Agent Systems - The Agent-Oriented Software Engineering Handbook*. F.Bergenti,MP Gleizes,F. Zambonelli (Eds). Kwuler Academic Publishers.

Sirin, E. Parsia, B. Wu, D. Hendler, J (2004). Filtering and selecting semantic Web services with interactive composition techniques. *IEEE Intelligent Systems*. 19:42-49.

Smeaton, A. and Quigley, I (1996). Experiment on Using Semantic Distance Between Words in Image Caption Retrieval. 19th International Conference on Research and Development in Information Retrieval SIGIR'96, Zurich, Switzerland.

Smith R.G. and Davis R (1981). Frameworks for Cooperation in Distributed Problem Solving. *IEEE Transactions on Systems, Man, and Cybernetics.* Vol. SMC-11, No. 1, pages 61–70.

Snell, J. (2001). Introducing the Web Services Flow Language. Accessed online at: http://www.ibm.com/developerworks/library/ws-ref4/. Last accessed: 10th November, 2008.

Sotomayor, B. & Childers, L. (2006). Globus Toolkit 4: Programming Java Services. Morgan Kauffman publishers.

Stone, P. & Veloso, M. (2000). Multiagent systems: A Survey from a Machine Learning Perspective (Tech. Rep. CMU-CS-97-193). Pittsburgh, PA: Carnegie Mellon University, School of Computer Science.

Sun, X. and Blateky, A. R (2004). Middleware: the key to next generation computing. *Journal of parallel and distributed computing*. Vol 64. pp 689-691.

Tanenbaum, A.S. Van Steen, M. (2007) Distributed Systems: Principles and Paradigms. Pearson Prentice Hall publishers. ISBN: 0-13-239227-5

Taverna (2004). URL: http://taverna.sourceforge.net/. Last accessed date: 10th June, 2009.

Thatte, S. (2001). XLANG Web Services for Business Process design. Online at: http://www.ebpml.org/xlang.htm.

Tidwell (2001). UDDI4J: Matchmaking for Web services. Online at: http://www.ibm.com/developerworks/library/ws-uddi4j.html. Last Accessed: 15th April, 2010

Triana (2003). URL: http://www.trianacode.org/. Last accessed date: 10th June, 2009.

Tuecke, S. Czajkowski, K. Foster, I. Frey, J. Graham, S. Kesselman, C. and Jennings N.R. (2002) Grid Service Specification, Presented at GGF4, February, 2002. Online at: http://www.globus.org/research/papers.html

Turner, K. (2000). Communication Representation Employing Systematic Specification. Online at: http://www.cs.stir.ac.uk/~kjt/research/cress.html.

Turner, K. J. & Tan, K. L. (2007). A rigorous approach to orchestrating grid services. *Computer Networks* , 51, 4421–4441.

UDDI (2002). Universal Description, Discovery, and Integration. Online: http://www.oasis-open.org/specs/index.php#uddi. Last Accessed: 15th April, 2010

Verity, J. Woods, D. Pattison, K. Adhikari, R. Cameron, D. Berr, J (2008). The Service Grid. In *International Research Forum 2008*. Woods D and Cameron D (Eds). pp 79-136. Evolved Technologist Press.

Verma, D.C (2004). Legitimate Applications of Peer-to-Peer Networks. Carlifonia: John Wiley & Sons.

W3C (2002a). Web Service Choreography Interface (WSCI) 1.0. Online at: http://www.w3.org/TR/wsci/. Date last accessed: 10th November, 2008.

W3C (2002b), http://www.w3.org. Last Accessed: 15th April, 2010

Wang, L. (2007). Toward Agent-Based Grid Computing. In Lin H (Ed.), Architectural design of multi-agent systems : technologies and techniques (pp 173-188). IGI Global publishing.

Weiss, G. (1999). Multiagent Systems: A modern approach to distributed artificial intelligence. The MIT Press.

Weske, M. (2008). Introduction. In: Semantic Service Provisioning. Dominik, K. Peter, T. Steffen, S. Mathias, W. (Eds.). Springer Publisher. ISBN 978-3-540-78616-0

WFMC (1995). Workflow Management Coalition (WFMC) reference model. Online at: http://www.aiai.ad.ac.uk/WFMC/. *Last accessed date:* May 10, 2009

Woods, D. Pattison, K. Cameron, D. (2009). The Web-based Service Industry. In: International Researcher Forum 2008. Woods D; Cameron D (Eds). Evolved Technologies Press. New York. ISBN: 978-0-9789218-8-0.

Woolridge, M. & Jennings, N.R. (1995). Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2), 115-152.

WSDL2Java (2000). Online at: http://ws.apache.org/axis/java/user-guide.htm. Last Accessed: 15th April, 2010

WSDL4J (2005). Online at: http://sourceforge.net/projects/wsdl4j. Last Accessed: 15th April, 2010

WSFL (2002). URL: http://www3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf. Last accessed date: June 10th, 2009.

Zeng, L. Benatallah, B. Lei, H. Ngu, A.H.H. Flaxer, D. Chang, H. (2003). Flexible composition of enterprise Web Services. Electronic Markets - Web Services, 13.

Zhang, L. Li, J. and Lam, H. (2004). Toward a Business Process Grid for Utility Computing, *IT Professional* 6: 62-64.

Zhang, L. Zhang, J. and Cai, H. (2007). Services Computing. Co-published by Tsinghua University Press and Springer- Verlag. ISBN: 978-7-302-15075-6.