Hasso–Plattner–Institut fuer Softwaresystemtechnik
an der Universitaet Potsdam

Temporary Binding for Dynamic Middleware Construction and Web Services Composition

Dissertation

zur Erlangung des akademischen Grades
"Doctor rerum naturalium"
(Dr. rer. nat.)
am Fachgebiet Internet Technologien und Systeme

eingereicht an der
Mathematisch–Naturwissenschaftlichen Fakultaet
der Universitaet Potsdam

von
Wanjun Huang

Potsdam, July 3, 2006

Acknowledgments

First and foremost, I greatly appreciate my supervisor, Prof. Christoph Meinel, not only for his help to get the financial supporting with which I was able to come to Germany to start my Ph.D programme, but also for his encouragement and guidance throughout my studies, from which I learnt the skills to be a researcher and how to work in a team environment. In addition, special thanks are given to Prof. Christoph Meinel for his contribution on the education and cooperation with my alma mater. I still remember the vivid scene when I was receiving the scholarship letter from his hands in a ceremony of cooperation held in Beijing.

I additionally would like to thank my former colleagues in Institute for Telematics and University of Trier, and current colleagues in Hasso-Plattner-Institut, for their helps as well as the many discussions in topics about research and life in general. Especially, I would further like to thank Dirk for his help to translate the abstract of my dissertation into German, and thank Debbie, Raveendra, Xinhua and Long for their helps to read partial chapters of my dissertation.

The special appreciation which should not be forgotten is given to my girlfriend Min for her love and encouragement. The happy time we spent let me easier and more relaxed, especially when I was struggling in my hard research work. I should also express my appreciation to my numerous Chinese colleagues and friends. Lots of parties we held often gave me a hallucination that I was back in China.

Finally but not least, I would like to thank my family for their loves and continuous supporting in my studies and life.

Abstract

With increasing number of applications in Internet and mobile environments, distributed software systems are demanded to be more powerful and flexible, especially in terms of dynamism and security. This dissertation describes my work concerning three aspects: dynamic reconfiguration of component software, security control on middleware applications, and web services dynamic composition.

Firstly, I proposed a technology named Routing Based Workflow (RBW) to model the execution and management of collaborative components and realize temporary binding for component instances. The temporary binding means component instances are temporarily loaded into a created execution environment to execute their functions, and then are released to their repository after executions. The temporary binding allows to create an idle execution environment for all collaborative components, on which the change operations can be immediately carried out. The changes on execution environment will result in a new collaboration of all involved components, and also greatly simplifies the classical issues arising from dynamic changes, such as consistency preserving etc.

To demonstrate the feasibility of RBW, I created a dynamic secure middleware system - the Smart Data Server Version 3.0 (SDS3). In SDS3, an open source implementation of CORBA is adopted and modified as the communication infrastructure, and three secure components managed by RBW, are created to enhance the security on the access of deployed applications. SDS3 offers multi-level security control on its applications from strategy control to application-specific detail control. For the management by RBW, the strategy control of SDS3 applications could be dynamically changed by reorganizing the collaboration of the three secure components.

In addition, I created the Dynamic Services Composer (DSC) based on Apache open source projects, Apache Axis and WSIF. In DSC, RBW is employed to model the interaction and collaboration of web services and to enable the dynamic changes on the flow structure of web services.

Finally, overall performance tests were made to evaluate the efficiency of the developed RBW and SDS3. The results demonstrated that temporary binding of component instances makes slight impacts on the execution efficiency of components, and the blackout time arising from dynamic changes can be extremely reduced in any applications.

Zusammenfassung

Heutige Softwareanwendungen fuer das Internet und den mobilen Einsatz erfordern bezueglich Funktionalitaet und Sicherheit immer leistungsstaerkere verteilte Softwaresysteme. Diese Dissertation befasst sich mit der dynamischen Rekonfiguration von Komponentensoftware, Sicherheitskontrolle von Middlewareanwendungen und der dynamischen Komposition von Web Services.

Zuerst wird eine Routing Based Workflow (RBW) Technologie vorgestellt, welche die Ausfuehrung und das Management von kollaborierenden Komponenten modelliert, sowie fuer die Realisierung einer temporaeren Anbindung von Komponenteninstanzen zustaendig ist. D.h., Komponenteninstanzen werden zur Ausfuehrung ihrer Funktionalitaet temporaer in eine geschaffene Ausfuehrungsumgebung geladen und nach Beendigung wieder freigegeben. Die temporaere Anbindung erlaubt das Erstellen einer Ausfuehrungsumgebung, in der Rekonfigurationen unmittelbar vollzogen werden koennen. Aenderungen der Ausfuehrungsumgebung haben neue Kollaborations-Beziehungen der Komponenten zufolge und vereinfachen stark die Schwierigkeiten wie z.B. Konsistenzerhaltung, die mit dynamischen Aenderungen verbunden sind.

Um die Durchfuehrbarkeit von RBW zu demonstrieren, wurde ein dynamisches, sicheres Middleware System erstellt - der Smart Data Server, Version 3 (SDS3). Bei SDS3 kommt eine Open Source Softwareimplementierung von CORBA zum Einsatz, die modifiziert als Kommunikationsinfrasturkutur genutzt wird. Zudem wurden drei Sicherheitskomponenten erstellt, die von RBW verwaltet werden und die Sicherheit beim Zugriff auf die eingesetzten Anwendungen erhoehen. SDS3 bietet den Anwendungen Sicherheitskontrollfunktionen auf verschiedenen Ebenen, angefangen von einer Strategiekontrolle bis zu anwendungsspezifischen Kontrollfunktionen. Mittels RBW kann die Strategiekontrolle des SDS3 dynamisch durch Reorganisation von Kollabortions-Beziehungen zwischen den Sicherheitskomponenten angepasst werden.

Neben diesem System wurde der Dynamic Service Composer (DSC) implementiert, welcher auf den Apache Open Source Projekten Apache Axis und WSIF basiert. Im DSC wird RBW eingesetzt, um die Interaktion und Zusammenarbeit von Web Services zu modellieren sowie dynamische Aenderungen der Flussstruktur von Web Services zu ermoeglichen.

Nach der Implementierung wurden Performance-Tests bezueglich RBW und SDS3 durchgefuehrt. Die Ergebnisse der Tests zeigen, dass eine temporoere Anbindung von Komponenteninstanzen nur einen geringen Einfluss auf die Ausfuehrungseffizienz von Komponeten hat. Ausserdem bestaetigen die Testergebnisse, dass die mit der dynamischen Rekonfiguration verbundene Ausfallzeit extrem niedrig ist.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The continuous emergence of new requirements for business software calls for the urgent need of developing large scale distributed systems, featured by easy integration, powerful functionality and high flexibility to adapt to the varying application environments. This chapter presents an introduction on three research issues involved in my work: dynamic reconfiguration on distributed system, security control on middleware applications and web services dynamic composition. Also covered are the contributions and the organizational structure of the present research.

1.1 Dynamic Reconfiguration

Dynamic reconfiguration is the ability to change the structure of a software system whilst the applications continue to be accepted and executed. The change operations could be, for example, adding a component, removing a component, replacing a component or reorganizing the collaboration relation of components. There are two typical application scenarios in which dynamic reconfiguration is strongly demanded. The first scenario is long-time running distributed system supporting continuous services in which a break of services or shutdown of system is not tolerable. So the dynamic reconfiguration on the software system is compulsory to update new version of components, add new functional components or remove broken components etc. Another application scenario demanding dynamic reconfiguration is reactive systems, such as mobile computing systems, in which different components are demanded to be flexibly reorganized to adapt to the varying application environment. Considering an application example of mobile computing, a sale manager is participating a high quality video conference with his customers using a mobile device in a train. With the moving of the train the signal bandwidth in new area becomes weak, the high quality video conference has to be switched to low quality video conference or even audio conference to enable the conversation continuous. In this case, the mobile application server has to detect the changes in its environment and dynamically make the reconfiguration on the system.

In [25] one kind of dynamic reconfiguration was distinguished as programmed reconfiguration, in which the changes are defined at the design time and are automati-

cally triggered by the system when the defined conditions are met. In this dissertation, the dynamic reconfiguration is only referred to the unplanned dynamic reconfiguration that is triggered by administrators or third-part programs at unpredictable time whilst the server is still running. A perfect dynamic reconfiguration should be capable of completing its execution within an expected short time and make no impact on the rest part of the system. Namely, all other unchanged components of the system should be able to function properly during the period of reconfiguration. However, the affected components may already be in the process of functioning when the change instruction arrives, so the dynamic reconfiguration is unavoidably an intrusive process for the whole system. The problems arising from dynamic reconfiguration could be summarized and classified as follows [66]:

Structure Integrity - Component IO Behaviors

The structure integrity aims to ensure the accuracy of the reconfigured system in terms of component IO behaviors. The reconfiguration approach has to specify the correct communication path between the changed components and the rest of the system so that the reconfigured system can continue seamlessly to provide application services. Regarding the replacement or update of components, the structure integrity is much easier to maintain. The new versions of components just have to keep the same IO behaviors of old components to keep compatibility. For the change operations, like adding new components or removing old components etc., a new communication path for each component has to be reestablished between itself and the surrounding components. The changes in the collaboration relation of components should always be transparent to the applications deployed in the system, namely the applications do not have to make any changes during the reconfiguration. However, the invocation method to access the applications might be changed after the reconfiguration, indicating a potential demand for reconfiguration. For instance, a secure component is required to be inserted into a system to enhance the security on the invocation of its applications.

Consistency Preserving - Change Process

During the period of reconfiguration, the affected components have to deal with two kinds of tasks: the functional tasks for ongoing application invocations and the reconfiguration tasks to make changes on itself. To guarantee the consistency between the two kinds of operations, the affected components have to be ensured a smooth transition between the two states: the state before reconfiguration and the state after reconfiguration. As a result, the reconfiguration has to start only when the system is in a safe state, viz no ongoing processing on the affected components. For the system supporting continuous services, there is no guarantee that the system will automatically go into a safe state when the reconfiguration instruction comes. In this case, reconfiguration algorithm will act as the role to drive the system into a safe state for successful reconfiguration. The existing approaches driving the system into a safe state could be roughly classified into two major categories:

- *Consistency through recovery* in which all affected components are forced by reconfiguration algorithm to abort any uncommitted transactions to drive the system into a safe state. After reconfiguration, the uncommitted transactions will be restored to complete the corresponding executions by the reconfigured system with the information recorded during abortions.

- *Consistency through avoidance* in which the abortion of ongoing processing is avoided and the affected components are gradually frozen into a safe state before reconfiguration. when the reconfiguration instruction comes, the reconfiguration algorithm checks every request and suspends the request whose execution will involve the affected components. So all affected components are able to go into a safe state in a limited time after the execution of ongoing request. Once all affected components reach a safe state, the reconfigurations are carried out and the suspected requests will be executed by the reconfigured system.

State Transferring - Component Inner State

Many attributes of the affected components, such as safety and aliveness properties etc., are rather conditional and critical to the smooth running of the system, which might call for a revision in its invariants in the subsequent requests executions. For example, there is an invariant used to record the frequency of requested execution. But after replacement, the invariant might be zeroed and thus leads to improper subsequent request executions. All kind of such cases have to be considered in reconfiguration. Therefore, reconfiguration algorithm much be able to identify the invariants affected and restore them.

Minimum Disruption - Change Performance

Disruption is a common phenomenon after dynamic reconfiguration, resulted from the temporary loss of efficiency of affected components. System disruption during reconfiguration can not completely be exempted as the influence of reconfiguration on the executions of components is ineluctable. An obvious solution of reducing system disruption is to minimize the time of execution of change to the smallest amount. This could be realized via an efficient reconfiguration algorithm to guarantee a seamless change: well preserved consistency and successful state transfer. During dynamic reconfiguration the time of reconfiguration is usually not equal to the time of disruption which is also called blackout time. In most cases, the blackout time is less than the reconfiguration time, since the affected components might still be in the process of request execution before a change operation is actually made.

1.2 Middleware Security

Secure middleware systems integrate security services into their platforms to provide a framework that enables users to easily create distributed applications and protect the sensitive applications from illegal accessing. Most security solutions for distributed

applications can be integrated into middleware systems. Typical security issues which are also closely related to my work, are introduced here and classified as follows:

Confidentiality and Integrity

In an enterprise application system, the sensitive data needs to be protected through a mechanism that prevents them from being read by intruder, called confidentiality, and being tampered during transit, called integrity.

Encryption is a widely used technique for data confidentiality. The modern encryption algorithms can be classified into two categories: symmetric key encryption in which the sender and the receiver of message share one key to encrypt and decrypt the message, and asymmetric key encryption, based on a pair of keys - a private key to encrypt data and a public key to decrypt data. Symmetric key ciphers are used extensively and ideal in connection-based situations where systems connect and exchange data and then disconnect. While the asymmetric encryption allows people with no pre-existing security arrangement to exchange message securely with the public key open to all people. Most commonly adopted symmetric encryption algorithms include Data Encryption Standard (DES), Advanced Encryption Standard (AES), etc.

The technology of checksum is probably the most basic form of integrity protection which produces a number based on the sum of all the bits to be compared against the target. Another prevalent tamper-detection technique is digital signature, which is based on an asymmetric algorithm. The digital signature is a scramble data encrypted with private key on the digest extracted from its original message. The well known algorithms are Digital Signature Algorithm (DSA), Message Digest 5 (MD5) etc.

Authentication

Many daily applications currently in use have taken the measurement of authentication to ensure proper usage, such as logining into a computer and checking emails etc.

The most commonly used technique for authentication is user ID/password pair. However, it is the least secure mechanism because of its simplicity. Another widely used technique is challenge/response protocol in which one entity proves its identity to another (the verifier) by demonstrating the knowledge of a secret known to both entities, without revealing the secret itself to the verifier.

The latest technology for authentication is X.509 Public Key Infrastructure (PKI) [61], based on the asymmetric algorithm. In PKI based systems, users ask for the Certificate Authority (CA) to acquire their private key and public key, already embedded into a certificate stored in a public repository. Anyone can verify a user by encrypting his signed certificate with the public key stored in the public repository. PKI based authentication technology is widely combined into the web oriented security protocol, such as SSL, SSH, etc.

Authorization

For sensitive applications it is not sufficient to identify a user and then give him a right to access all resources. A more effective means is to provide different level of services for different users, called authorization or privilege management.

The typical technique to manage privileges is to group users according to their levels. Group eases the task of privilege management since certain privileges could be assigned automatically to each user of the corresponding group. Typical application of grouping is the privilege management in Unix or Windows operating system.

Role based privilege management is another recently developed technique adopted in all kinds of applications. Different from grouping, role technique is intended to combine a collection of users and a collection of permissions. It is more reasonable and easier to manage the permission assigned to one role, and assign different roles to individuals. The example of permission could be a series of operations on certain object, like read, write, execute, delete etc.

Access Control

Access control strategies are often integrated with authorization management systems to provide a framework to control the access restriction of users on different resources. The earlier access control strategy employed in operating systems is Access Control List (ACL), which consists of a set of ACL entries each of which specifies the access right of individual user or group to a specific system object. Two other control mechanisms based on group are Discretionary Access Control (DAC) which permits user to grant or revoke access right to any of the objects under their control, and Mandatory Access Control (MAC) in which the protections are done by system administrators to restrict the access to objects according to the sensitivity of the information contained in the objects. The new appeared technology for access control is Role Based Access Control (RBAC) that provides administrators a more flexible way to regulate who can perform what actions, when, from where, in which order, and in some cases under what relational circumstances. The RBAC system enables users to carry out a broad range of authorized operations in wide applications. System administrators can control access at a level of abstraction which is reasonable to the way that enterprises typically conduct business.

In the popular middleware systems, such as CORBA, RMI/J2EE and DCOM/.NET etc., rich functionalities concerning security are provided to develop secure applications. CORBA integrates wide variety of security models to enable comprehensive security capabilities. As a meta-model, CORBA security specification has no implementation. The open source and commercial implementations of CORBA also realize partial security capabilities and features. Another two counterparts: .NET and J2EE have the similar architecture and security framework. .NET has the fundamental execution environment - Common Language Runtime (CRL) which has security system and policy setting to ensure the running time secure code verification. J2EE has the Java Virtual Machine (JVM) which has a security manager to ensure the secure invocation of applications. In addition, both .NET and J2EE provide a rich set of

security Application Programming Interfaces (APIs) for flexible and powerful security control on distributed applications. However, security solutions offered by traditional middleware systems are not flexible to be easily changed to meet the varying requirements. In addition, they are too complex and not suitable for the small or middle scale applications which also need powerful security controls.

1.3 Dynamic Services Composition

Service-Oriented Computing (SOC) is becoming a prominent computing paradigm that utilizes services as fundamental elements for distributed enterprise applications. Web service is a practical technology to fulfill the architecture of SOC. Web service is widely accepted and appears as a new distributed computing technology for it enables interoperability among all kinds of different traditional computing platforms, such as CORBA, DCOM and J2EE etc., and to easily wrap the large number of legacy projects. There are a set of closely related standards and specifications proposed as the core technologies to fulfill web services computing. Web Services Description Language (WSDL) is an XML format to describe web services as collections of network endpoints. Simple Object Access Protocol (SOAP) is a protocol for exchange of message information in a decentralized environment. Universal Description Discovery and Integration (UDDI) is a specification for technical model and standard interfaces for web service registries from which companies can publish their offered services and customers can look up their demanded services. On top of above core web service technologies, web services composition is explored to provide an open, standards-based approach for integrating web services to create more powerful composite services. Currently most of the plenty research works concerning web services composition can be categorized into the following aspects:

Composition Language

The composition language is a formal specification to define the service activities and specify how the services interaction is processed to reach the complete execution of a global composite service. Most work of composition language focuses on:

- The definition of basic service activities and relevant contexts to ensure the correct invocation and matching of web services;
- The regulation for service interaction and error handling to ensure correct service processes.

There are two kinds of composition languages or specifications. One is proposed and pushed forward by industrial companies, such as Web Service Flow Language (WSFL) proposed by IBM, XLANG proposed by Microsoft, Web Service Choreography Interface (WSCI) proposed by Sun, SAP et al, and BPEL4WS proposed by Microsoft, IBM et al etc. The other is spurred by non-profit international organization, such as OWL-S: Semantic Markup for Web Services which is proposed by academic researchers and submitted to W3C.

Services Enactment

The software system of web service enactment is responsible for managing the services process, among which the main issues focus on the following sub-items:

- *Infrastructure* which is constructed to compose an application server or integration server to enable effective enactment of web services processes.
- *Reliability*: for web services are loosely coupled, reliable process of services enactment is especially important for the applications in enterprise business.
- *Security*: web services are open to the Internet,so security mechanism are demanded to restrict the access of services to the authorized clients, which is indispensable for sensitive or business service applications.
- *Transaction*: the execution of a composite service succeeds only if all constituted sub-services are invoked successfully. Transaction mechanism is demanded to ensure all the sub-services successful or failed.

Automatic and Dynamic Composition

One key academic research effect on service composition is automatic discovery, selection and dynamic composition of web services. The dynamism of composition focuses on the dynamically selecting and changing the service providers, or dynamically changing the flow structure of composite services. The automatic and dynamic capabilities are related to the representation in composite language, but they depend more on the infrastructure and the framework of service enactment. Currently there are two kinds of approaches to reach flexible services processes: workflow technology based business services process and semantic web based services composition.

Performance Simulation and Monitor

The performance simulation, predication, evaluation and monitor on the web services execution is a quite new research field in which only few research work has been done. However, it has gained more and more attentions from the researchers as web services are currently applied in many practical enterprise applications. As constituted parts of composite services, web services may come from different departments, or even from different companies and organizations, which pose much challenges and enormous costs for estimating and testing a new business application based on composite services. This calls for a simulation system for composite services execution to model, simulate and even predict the performance of web services.

1.4 Contributions and Organizational Structure

Dissertation Contributions

The most important contribution of this dissertation lies in the proposal of a novel approach - Routing Based Workflow (RBW) for dynamic reconfiguration on component

software. The RBW models the collaborative components from schema specification to running state and realizes a concept of temporary binding for component instances. The temporary binding means the component instances are temporarily loaded into a created execution environment to execute their functions, and then released to their repository after executions. In comparison with other approaches for dynamic reconfiguration, the two prominent advantages of RBW lies in that:

1. It greatly simplifies the classical issues arising from dynamic changes, like consistency preserving. Based on the experimental data tested in advance, the reconfiguration time becomes predictable, and the blackout time could always be limited in an extremely small value in any applications.

2. It supports multi-solutions synchronously, which can be employed to develop software systems that provide personalized services.

The second contribution is a secure, dynamic middleware system - Smart Data Server Version 3.0 (SDS3) created on top of CORBA. The SDS3 not only demonstrates the feasibility of RBW, but also provides a secure framework to enable multi-level security control on its deployed applications. The detail control allows to specify the application-specific or operation level access control by policy configuration. The strategy control allows to specify application-independent security strategy by reorganizing the secure components managed by RBW. For the dynamism inherited from RBW, the security strategy of the system is able to be dynamically changed during runtime.

The third contribution is the Dynamic Services Composer (DSC) which is created on top of Apache Axis and employs the technology of Apache WSIF and our proposed RBW. The DSC demonstrates that the RBW can also be used to compose web services to easily create more powerful business services. A distinctive advantage of DSC is that it enables the dynamic change of the flow structure of composite services.

Organizational Structure

This dissertation is organized as follows:

- In chapter 1 I introduce three research issues involved in this dissertation: dynamic reconfiguration on distributed systems, security control on middleware applications and web services dynamic composition.

- In chapter 2 the basic distributed technologies involved in my research work are introduced: object oriented distributed computing, web services technologies, and workflow management.

- Chapter 3 presents the introduction and analysis of the evolution of dynamism in component oriented software system from configuration to dynamic reconfiguration and computational reflection. Also presented is a more flexible capability that is realized in my work: multi-solutions synchronously supporting.

- Chapter 4 presents the focus of this dissertation: Routing Based Workflow (RBW) covering its structure, modeling, execution, management, schema, and extension for distributed components.
- Chapter 5 gives the first case study of RBW: Smart Data Server Version 3.0 (SDS3) covering topics of how to create SDS3 on top of CORBA and how to design the secure components managed by RBW.
- Chapter 6 gives the second case study of RBW: Dynamic Services Composer (DSC). Firstly I explain how the RBW is used to model service composition, and then introduce the system architecture which is based on the Apache Axis and WSIF.
- Chapter 7 presents an overall performance tests of the developed RBW and SDS3. Also presented is an analysis on the execution efficiency and reconfiguration time.
- Chapter 8 is an introduction of the related works concerning three issues. In addition to a summary of different approaches, evaluation and comparison between them are also provided.
- Chapter 9, the conclusion chapter, presents a summary and analysis of the present work together with a profile of perspective future work.

Chapter 2

Concerned Distributed Computing Technologies

In this chapter, the distributed computing technologies which are closely related to my work are presented. The first parts are object oriented distributed computing technologies which are involved in my first case study, a middleware system. The second parts are web service technologies which are the fundamental technologies for web service composition. The third parts of introduction go to the workflow management which is related with the core of my work, routing based workflow.

2.1 Object Oriented Distributed Computing

Like Remote Procedure Call(RPC), object oriented distributed computing offers synchronous, typed communication between components of distributed program. The widely accepted solutions are CORBA from OMG, DCOM/.NET from Microsoft, and Java RMI/J2EE from Sun etc., which are introduced as follows.

2.1.1 Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) is an open distributed object computing infrastructure being standardized by the Object Management Group (OMG) [40]. CORBA automates many common network programming tasks such as object registration and activation; request de-multiplexing, framing and error-handling; parameter marshalling and de-marshalling; and operation dispatching etc. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language can interoperate with another CORBA-based program from the same or another vendor, on the same or another computer, operating system and network.

CORBA applications are composed of objects, individual units of running software that combine functionality and data. Typically there are many instances of an object of a single type. For each object type, it is defined in the form of interface. The interface is the syntax part of the contract that the server object offers to the clients.

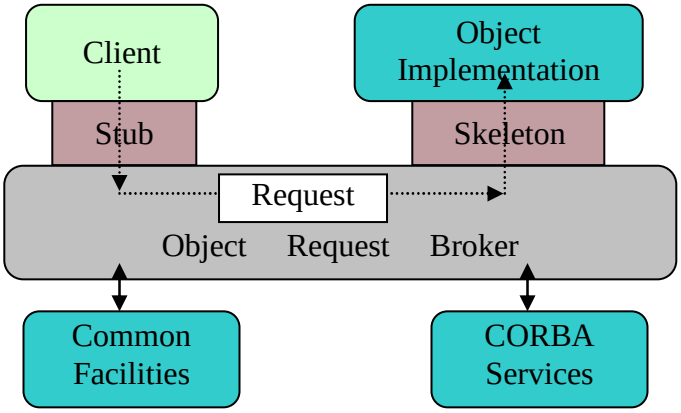

Figure 2.1: Request Invocation of CORBA

As illustrated in Figure 2.1, any client that wants to invoke an operation on the object must use this interface to specify the operation it wants to perform, and use its stub to marshal the arguments that it sends. When the invocation reaches the target object, the same interface definition is used there by its skeleton to un-marshal the arguments so that the object can perform the requested operation. The separation of interface from implementation is the essence of CORBA to enable interoperability. The interface to each object is defined very strictly. In contrast, the implementation of an object, its executable code and data, is hidden from the rest of system and behind a boundary that client may not cross. In CORBA, every object instance has its own unique object reference with which clients direct their invocations.

CORBA Architecture

The Figure 2.2 illustrates the architecture of CORBA, in which the individual core components are explained further as follows:

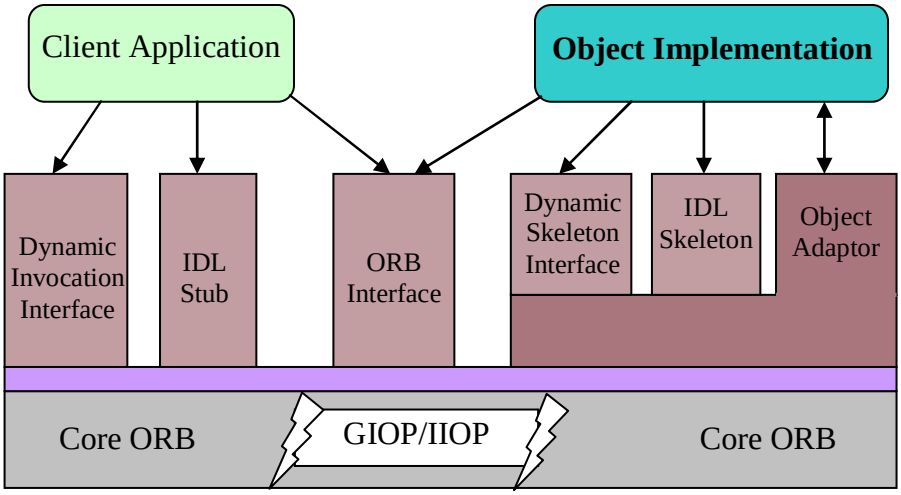

Figure 2.2: CORBA Architecture

Client Application This is the program entity that invokes an operation on an object implementation. Ordinarily, a client calls a routine in its program that performs the invocation and returns when the operation is complete.

Dynamic Invocation Interface (DII) The Dynamic Invocation Interface (DII) allows dynamic creation and invocation of request to objects. Using the DII mechanism, an object is accessed by a call to the ORB in which the object, method and parameters are specified. It is the client's responsibility to specify the types of the parameters and the expected results.

IDL Stub and Skeleton IDL stubs and skeletons serve as the "glue" between clients and server applications. This is the static invocation interface, representing a language mapping between the client language and the ORB implementation. The transformation between CORBA IDL definitions and the target programming language is automated by an IDL compiler.

Core Object Request Broker (ORB) The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies decoupling the clients from the details of the method invocations. When a client invoke an operation, the ORB is responsible for finding the object implementation, delivering the request to the object, and returning any response to the caller.

GIOP/IIOP CORBA uses the General Inter-ORB Protocol (GIOP)to define the format of messages and uses the Internet Inter-ORB Protocol (IIOP) to map GIOP messages to TCP/IP messages. IIOP allows ORBs to communicate with each other and enables them to use the Internet protocols for communication of distributed objects.

ORB Interface The ORB interface allows functions of the ORB to be accessed directly by the client code. This interface provides only a few operations, such as stringifying an object reference, that are shared by both the client side and the implementation side of CORBA architecture.

Dynamic Skeleton Interface (DSI) This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation without the compile-time knowledge of the type of the object. The client invoking a request has no idea whether the implementation is using the type-specific IDL skeleton or is using the dynamic skeleton.

Object Adapter This assists the ORB delivering requests to the object and activating the object. More importantly, an object adapter associates object implementations with the ORB. The Object Adapter sits on top of the ORB Core communication services, accepting request for service on behalf of the server's object. The object implementation accesses most ORB services through the Object Adapter. CORBA specifies two candidate adapters called the Basic Object Adapter (BOA) and Portable Object Adapter (POA).

Object Implementation An object implementation provides the actual state and functionality of an object. Besides defining the method for the operations themselves, an implementation will usually define procedures for activating and deactivating objects and will use other object or non-object facilities to make the object state persistent.

2.1.2 Microsoft Technologies

Distributed Component Object Model

The Microsoft Distributed Component Object Model (DCOM) [24] is a protocol that enables software components to communicate directly over a network. DCOM extends the Component Object Model (COM) [13] to support communication among objects on different computers. The communication protocol of DCOM, Object Remote Procedure Call (ORPC), is based on Open Software Foundation's DCE-RPC specification. With DCOM, business applications can be distributed at any locations that make the most sense to customers and application provider can focus on the real business instead of low level details of network protocol. Key features of DCOM are introduced in more detail as follows:

Component Interoperability and Reusability The DCOM is designed to allow two or more components to easily cooperate with one another, even if they were written by different vendors in different programming languages, or if they are deployed on different machines running different operating system. The COM defines a completely standardized mechanism for creating objects and for clients and objects to communicate. These mechanisms are independent of application and the programming language, and the key technology is a binary interoperability standard. DCOM makes distinctly separation for the interface from its implementation. Interface is a semantic contractual way for component to expose its service behavior. The reusability of component depends upon the interface and not the exact implementation.

Location Independence DCOM completely hides the location of a component, the client software don't care about whether server is in the same process as the client or in a machine halfway around the world. In all cases, the way the client connects to a component and calls the component's method is identical. Not only does DCOM require no changes to the source code, it does not even require the recompilation of program. Behind the unique invocation way of client, DCOM server components can be classified into three types: in-process, local process and remote process. In-process component is based on Dynamic Link Library (DLL), can be loaded and accessed directly. Local process and remote process components are implemented as stand-alone executable module that can be accessed via proxy and stub.

Language Neutrality DCOM is completely a binary and language independent specification. Virtually any language can be used to create COM components,

and those components can be used for even more language, such as Java, Microsoft Visual C++, Microsoft Basic, Delphi etc. In the language neutrality COM Interface Definition Language (IDL) plays a fundamental role to specify the interface behavior of component and separate interface from its implementation. When a designer creates an interface, that designer usually defines it using an IDL. From this definition an IDL compiler can generate header files for programming languages such that application can use that interface create proxy and stub object to enable object oriented RPC across network.

Uniform Data Transfer DCOM provides interfaces for dealing with uniform storage of object which enable to data exchange between applications. The persistent storage technology on top of COM is Uniform Data Transfer, which provides the functionality to represent all data transfers. Just as the meaning of the name "Uniform", this technology separates all the common exchange operation from what is called transfer protocol. This uniformity not only reduces the code necessary to source or consume data, but also greatly simplifies the code needed to work with the protocol itself.

.NET Platform

.NET is the Microsoft web services strategy to connect information, people, system and device through software [22]. .NET connected solution enables business to integrate their systems more rapidly and in a more agile manner and help them realize the promise the promise of information anytime, anywhere, on any device. The fundamental infrastructure of .NET platform is .NET framework [86] which constructs the general runtime environment closely associated with the operating system. Microsoft also provides a serial of tools and servers to build and deploy enterprise application, such as Visual Studio.NET to build application, Internet Information Server (IIS)and Application Centre Server to manage and deploy the business applications etc. The .NET framework is an integral windows component for building and running the next generation of software applications and web services. It includes the Common Language Runtime (CRL) environment, a just-in-time compiler and a set of operating system libraries packaged with .NET components model.

Common Language Runtime The most important component of the .NET framework is the Common Language Runtime (CRL). The CLR manages and executes code written in .NET languages and is the basis of the .NET architecture, similar to Java Virtual Machine (JVM) of J2EE platform. The CLR is the engine that drives the execution of every .NET application. It activates objects, performs security checks on them, lays them out in memory, and execute them. As illustrated in Figure 2.3, the CLR consists of the Just-In-Time compiler (JIT) that compiles Common Intermediate Language (CIL) to native object code, the garbage collector, the Common Type System (CTS) and the exception handling machinery.

The CLR is language neutral, which means it can run code written in C#, Visual Basic.NET, or any other language as long as the language can be firstly compiled to

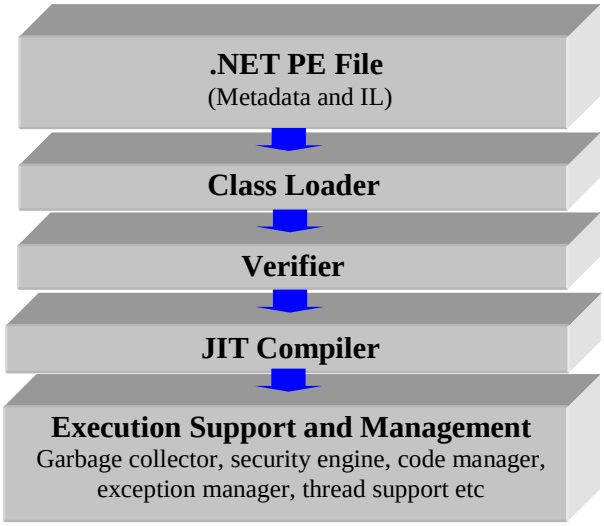

Figure 2.3: Structure of Common Language Runtime

the byte stream format that the CLR expects. When compiling to managed code, the language-specific compiler translates the source code into Microsoft Intermediate Language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code. MSIL includes abstract instructions for loading, storing, initializing, and invoking methods on objects, as well as abstract instructions for arithmetic, control flow, direct memory access, exception handling and other operations. The compiled MSIL is placed into an EXE or DLL file, called a Portable Executable (PE) file. Before code can be run, the MSIL in the PE must be converted to CPU-specific code, usually by a JIT compiler. During execution, managed code receives services such as garbage collection, security, cross language debugging support and enhanced deployment from CLR.

Enterprise Application Development The general model of distributed enterprise application with .NET platform is depicted in Figure 2.4. The technologies concerning enterprise applications developing and deploying can be classified into three tiers: presentation tier, business tier and database tier.

ASP.NET takes charge of the task of presentation tier in .NET platform. It is a technology for creating dynamic web applications and web services that can be easily accessed via web browser or .NET client systems.

Business tier assembles core technologies of .NET platform where VisualStudio.NET provide a integrated developing environment to build application and services, and COM+ is used to model the business services. Additionally, a series of .NET enterprise servers, such as Application Centre Server, Host Integration Server, BizTalk Server etc, are offered to deploy and manage the middle tier applications.

For the third tier of database, .NET provides a high performance enterprise database SQL Server to store enterprise data, and ADO.NET provides a rich set of API for conveniently accessing SQL Server or ADO compatible database.

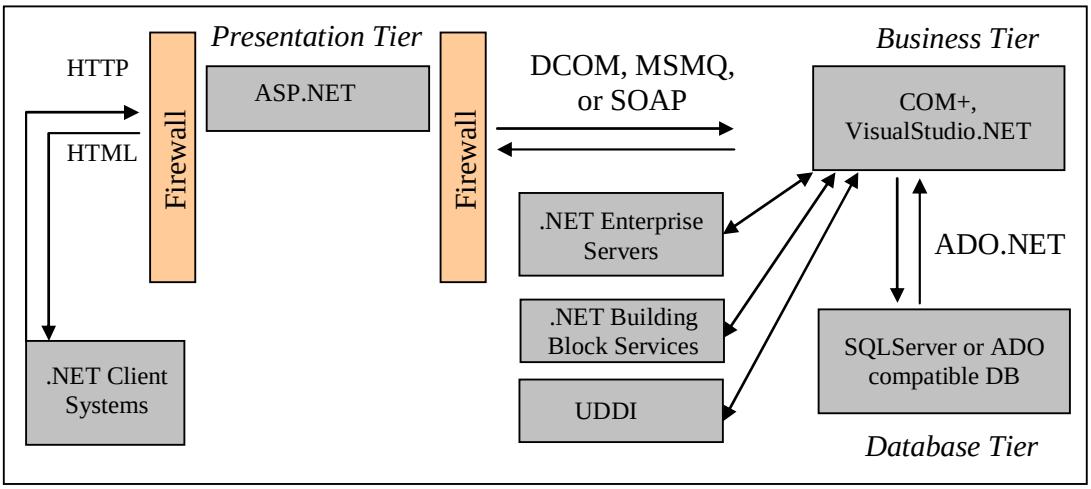

Figure 2.4: E-Commerce Architecture of .NET Platform

2.1.3 Java Technologies

Java Remote Method Invocation

Java Remote Method Invocation (RMI) is an object oriented distributed computing technology provided by Sun Microsystems [63]. RMI allows applications to call object methods located remotely. Unlike other systems for remote execution which require that only simple data types or defined structures be passed to and from methods, RMI allows any Java object type to be used and allows new object types to be loaded dynamically as required. Java RMI is not language neutral, but a Java-only distributed object solution where objects have to be implemented in Java. For RMI is built into the core Java environment since the release of JDK version 1.1, there is no need to install any other software or tools to enable the execution of RMI based application in Java environment, and the integration of remote object facilities into Java application is almost seamless.

Framework As indicated in Figure 2.5, the framework of Java RMI gives an overview that how the remote object can be invoked. There are three layers that comprise the basic remote object communication facilities in RMI:

1. *Stub/Skeleton Layer*, which provides the interface that client and server application use to interact with each other, and delegates the invocation operations;

2. *Remote Reference Layer*, which handles the creation and management of remote object references;

3. *Transport Protocol Layer*, which is the binary data protocol that sends remote object request and response over the wire;

In RMI based environment, a client makes a request of a remote object by calling a method on a stub object that shares the same interface of remote object. If

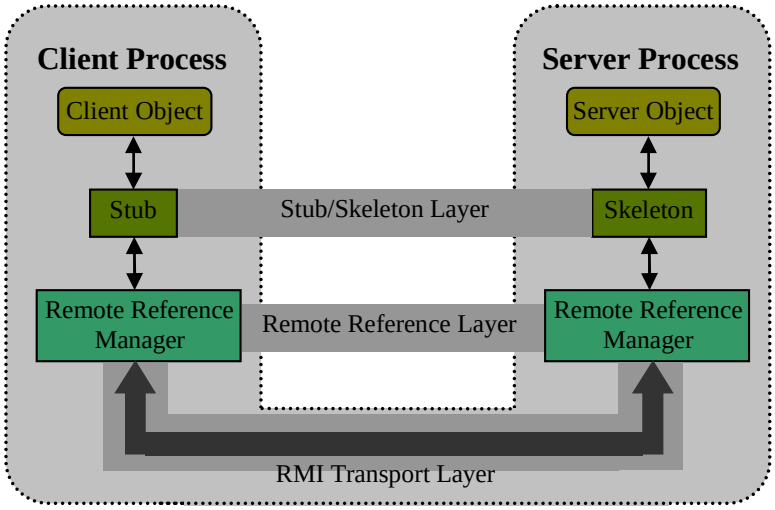

Figure 2.5: Framework of Java RMI

the marshalling of method argument succeeds, the remote reference layer converts the client request into low level communication oriented bytes stream that will be transported to server via wire communication protocol. On the server, the server side remote reference layer receives the transport level request and converts it into a request for the skeleton. The Skeleton then delegate the task of invocation, and enable the invocation seems to come from remote client.

Client Stubs and Server Skeletons Stub and Skeleton are respectively the proxy of remote object in client side and server host. The Stub delegates the client request to server and retrieve invocation result through transport protocol (JRMP or IIOP over RMI), and the Skeleton intercepts the request to enable the invocation on object implementation. With client Stubs and server Skeletons, the invocation on remote object over distributed network looks invocation on local object. Once the object interface is defined and derived by a server implementation, the client Stub and server Skeleton can created by RMI compiler, rmic. In Java 2 Platform, the Skeleton for specific object is not necessary for the general skeleton can be created via Java reflection technology.

The RMI Registry In RMI the registry serves the role of the Object Manager and Naming Service for the distributed object system. The registry is only required to be running on the server of a remote object, and clients of the object use the RMI package to communicate with remote registry to look up objects on the server. Once the registry is running on the server, the object implementation can be registered by name using Naming interface, java.rmi.Naming. A registered object can then be located by a client using lookup method on Naming services.

J2EE Platform

The Java 2 Platform of Enterprise Edition (J2EE) [64] defines the standard for developing multi-tiers enterprise applications. The J2EE simplifies enterprise applications by standardized, modular components, by providing a complete set of services and by handling many details of application behavior automatically. The primary J2EE technology components for creating enterprise applications are Java Servlet, Java Server Page (JSP) and Enterprise JavaBean (EJB). Additionally, a series of services or servers are also provided to support application development and deployment: Java Database Connectivity (JDBC) for platform and vendor independent accessing to SQL compliant database, Java Message Service (JMS) to provides a Java API for message queuing, publishing and subscribing, Java Transaction API (JTA) for distributed transaction management, Java Naming and Directory Interface (JNDI) etc. As shown in Figure 2.6, a J2EE enterprise application can be divided in four tiers: client application, Servlet and JSP based presentation, EJB based business logic and JDBC based database. Beside of Sun, another two companies provides commercial implementation for J2EE specifications: Websphere from IBM Corporation and Weblogic from BEA System.

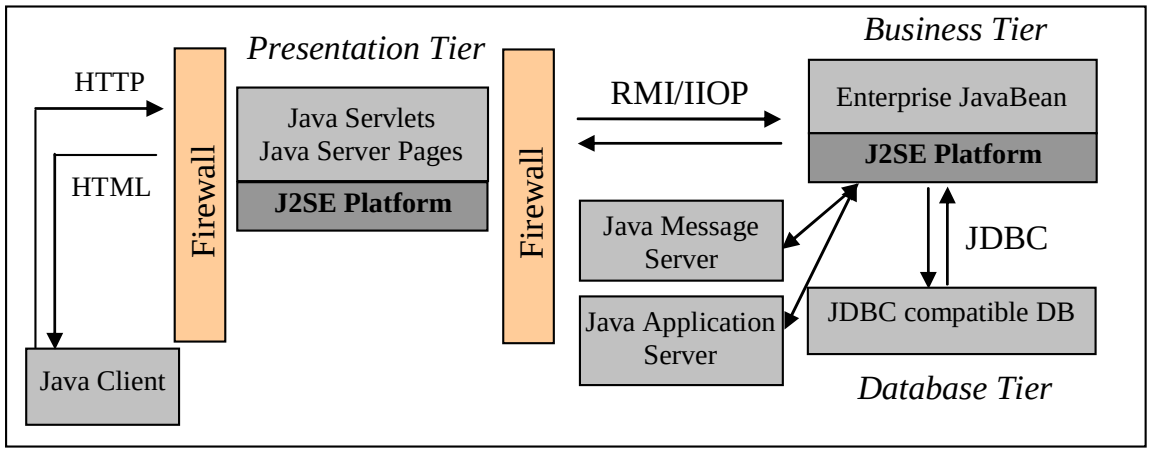

Figure 2.6: Enterprise Application Architecture of J2EE Platform

Java Servlet and JavaServer Pages The Sevelet is protocol and platform independent server side components, written in Java, which dynamically extend Java enabled servers. It provides a general framework for services built using the request-response paradigm. Its general use is to provide secure web-based access to data which is presented using HTML web page, interactively viewed or modified using dynamic web page generation techniques.

The JavaServer Page (JSP) technology provides a simplified, fast way to create web pages that display dynamically generated content. To eases and speeds the developing of dynamic web pages, a number of mechanisms are employed:

1. *Separating content generation from presentation*: JSP use HTML or XML tags

to design and format the results page, and use JSP tags or scriptlets to generate the dynamic content;

2. *Emphasizing reusable components*: JSP pages exploit reusable, cross-platform components, such as servlet, to perform the complex application processing;

3. *Simplifying page development with tags*: JSP technology encapsulates much of the functionality for dynamic content generation in easy-to-use, JSP-specific XML tags.

Enterprise JavaBeans Enterprise JavaBeans (EJB) is a specification for creating Java based, server side and reusable component framework for distributed applications [62]. The basic EJB architecture is composed of an EJB client, EJB server, EJB container and EJB. The client does not directly invoke methods of the EJB, and the container acts as an intermediary between the EJB and the client. EJB servers are analogous to the ORB in CORBA, and provide the system services, such as multiprocessing, load balancing, naming and transaction services etc. EJB run in a special environment called EJB container that manages every aspect of an enterprise bean at runtime, including remote accessing to the bean, security, persistence, transaction and pooling of resources. The enterprise bean focuses only on business logics and rules, while the container takes care of everything else.

The EJB specification defines three distinct types of enterprise beans: session beans, entity beans and message-driven beans. A session bean represents a business conversation with a single client and is not shared across clients. An entity bean is intended to represent the business logic for an entity existing in persistent storage, and allow shared access by multiple clients. A message driven bean is used for application to handle messages asynchronously, and act as a message listener in EJB container when it is instantiated.

2.2 Web Services Technologies

In the section 2.1 the classical solutions for distributed computing, such as CORBA, Java RMI etc., are introduced. However, those approaches have yielded the systems where the coupling between various components in a system is too tight to be effective for low-overhead, ubiquitous enterprise e-business over the Internet. The concept of Web Services is the next generation of e-business architecture web-oriented. The web services architecture [12], [5] describes principles for creating dynamic, loosely coupled systems based on services, and promotes significant decoupling and dynamic binding of components. All components in a system are services, in which they encapsulate functional behavior and publish a messaging API to other collaborating components on the network.

Web services architecture involves many layered and interrelated technology families, as depicted in Figure 2.7. The eXtensible Markup Language (XML) is the cornerstone of all web services technologies. The XML technology families involved in web services consist of XML specification, XML Schema description language and

XML Base specification. The basic standards of web services technologies are Simple Object Access Protocol (SOAP) that sends and receives messages over the network in a standard format, Web Services Description Language (WSDL) that describes the provided functionalities of services and how these services can be accessed and invoked, Universal Description, Discovery and Integration (UDDI) that describes an online electronic registry to serves as electronic yellow pages and to provide an information structure where various business vendors register their services. More advanced processing models and specifications of web services are being continuously proposed, such as Web Services Addressing (WS-Addressing) provides a transport - neutral mechanisms to enables messaging systems to support message transmission through networks, the specification of Web Services Choreography models business process that involves multiple different organizations and enables the information exchanging among these business services if they are properly coordinated.

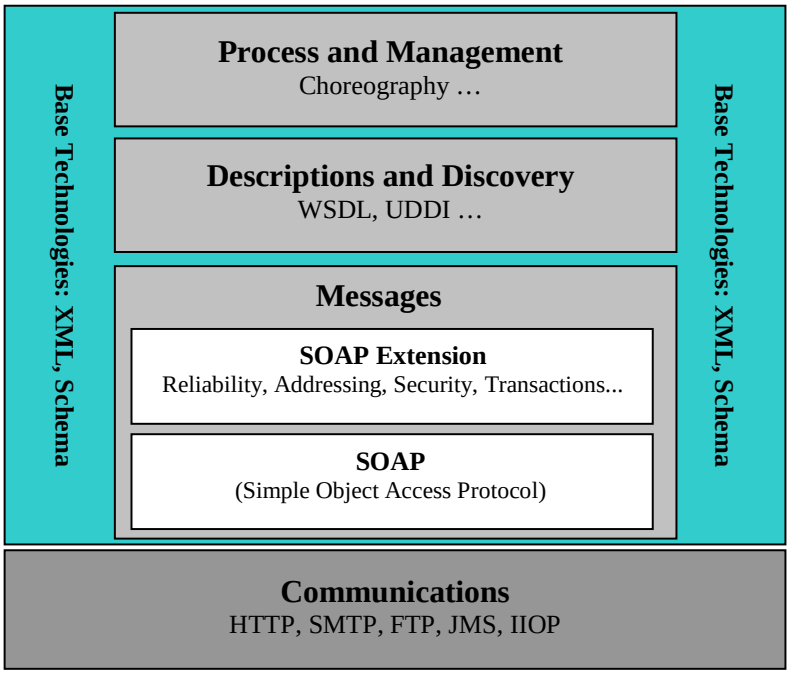

Figure 2.7: Diagram of Web Services Technologies

2.2.1 Web Services Description Language

Web Services Description Language (WSDL) [11] is an XML formatted language used to describe web services as collections of communication endpoints capable of exchanging messages that contains either document oriented or procedure oriented information. Currently, the version 1.1 is considered as the de-facto standard that gets industry-wide support. However, a new version 2.0 is being worked to be a recommendation endorsed by the W3C.

WSDL separate the description of the abstract functionality offered by a service from concrete details of service description that contains protocol binding and message

formats required to interact with the services listed in its directory. This separation supports the reuse of abstract definitions of WSDL elements that may be bound to multiple concrete protocols.

Abstract Definition

In WSDL the abstract definition gives a protocol and network address independent description for the provided functionalities, Operations, and the relevant invocation way, Messages. The abstract elements of WSDL are explained as follows:

- *Types* - a container for data type definitions using XML Schema based data type system.
- *Message* - an abstract, typed definition of the data being communicated and invocation way defined by message pattern.
- *Operation* - an abstract description of an action supported by the service.
- *Port Type* - an abstract set of operations supported by one or more endpoints, also called Interface in version 2.0

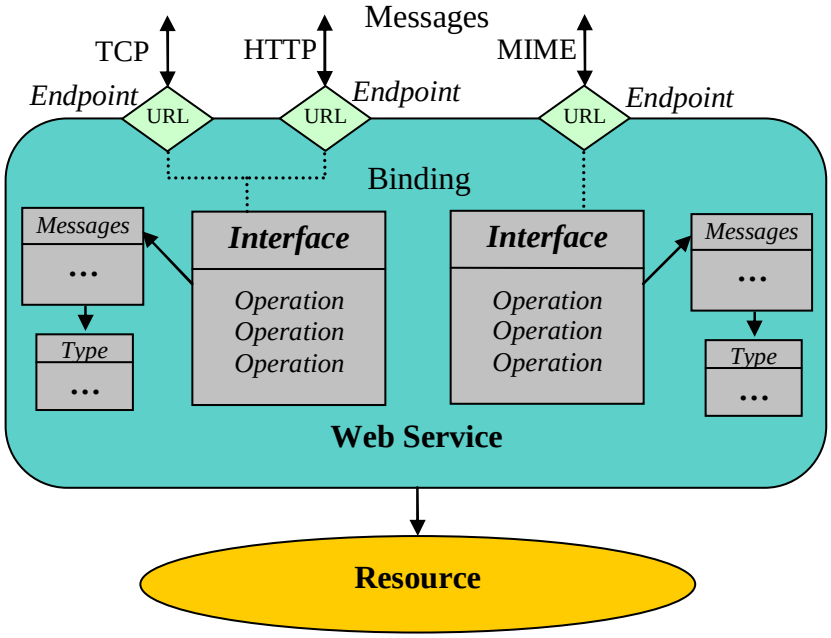

Figure 2.8: Relation Between Elements of WSDL

Concrete Definition

The concrete definitions of WSDL elements binding the abstract services to the specific protocol, such as SOAP, HTTP GET/POST, MIME etc, so that other web services know how to interact with them. The elements for concrete definition are: Binding, Port and Service.

- *Binding* - a concrete protocol and data format specification for a particular port type.
- *Port* - a single endpoint defined as a combination of binding and a network address.
- *Service* - a collection of related endpoints.

2.2.2 Universal Description, Discovery and Integration

The Universal Description, Discovery and Integration (UDDI) project is originally an industry initiative, the first truly cross-industry effort driven by all major platform and software vendors and e-business leaders [72], [71]. The UDDI specification defines a framework for describing services, discovering business and integrating business services using Internet. In another words, UDDI is a web-based distributed directory that enables businesses to list themselves on the Internet and discover each other, similar to a traditional phone book's yellow pages. UDDI specification is based on a set of industry internet standards, such as XML, WSDL and SOAP etc. Key technologies and functionalities offered by UDDI are introduced as the follows:

UDDI Data

UDDI presents an information model composed of instances of persistent data structure called entities. The key entity types are:

- *Business Entity*: Describes a business or organization that provides web services. Each entity contains its name, description, contact information, categories, identifier and an URL pointing to more information about the business.
- *Business Service*: Describes a collection of related web services offered by an organization. Each business entry contains name, description, categories and a list of references related to the service.
- *Binding Template*: Describe the technical information, such as web service URL etc, necessary to access a particular web service.
- *Technical Model*: Represents a reusable meta-data, such as a web service type, a protocol used by web services or a category system.

UDDI Services and API Sets

The specification presents services and API sets that standardize behavior and communication with or between implementations of UDDI for the purposes of manipulating UDDI data stored within those implementations. The API sets grouped into UDDI Node are UDDI Inquiry, Publication, Security, Custody Transfer, Subscription and Replication.

UDDI Nodes

A set of web services supporting at least one of the Node API sets is referred as a UDDI node. A UDDI node is a member of exactly one UDDI registry and supports interaction with UDDI data through one or more UDDI API sets. A UDDI node conceptually has access to and manipulates a complete logical copy of the UDDI data managed by the registry. Typically, UDDI replication occurs between UDDI nodes which reside on different systems in order to manifest this logical copy in the node.

UDDI Registries

One or more UDDI nodes may be combined to form a UDDI registry. The nodes in a UDDI registry collectively manage a particular set of UDDI data. A UDDI Registry is comprised of one or more UDDI nodes that collectively manage a well-defined set of UDDI data, which is typically supported by the use of UDDI replication. A registry must make a policy decision for each policy decision point. It may also be chosen to delegate policy decision to nodes.

2.2.3 Simple Object Access Protocol

The Simple Object Access Protocol (SOAP) is an XML based protocol that provides a simple and lightweight mechanism for exchanging structured and typed information between systems in a decentralized and distributed environment [65]. SOAP is platform and language independent and was originally intended and defined for use on top of HTTP to make SOAP more easily incorporated into Web-based applications, and other transport protocols, such as SMTP, MIME etc., can also be used. SOAP consists of three parts:

- *The SOAP envelope* construct defines an overall framework for expressing what is in a message, who should deal with it, how to process it and whether it is optional or mandatory.
- *The SAOP encoding rules* defines a serialization mechanism that can be used to exchange instances of application defined data types.
- *The SOAP RPC representation* defines a convention that can be used to represent remote procedure calls and responses.

SOAP messages are often combined to implement pattern of request/response. All SOAP messages are encoded in the form of XML document that consists of a mandatory SOAP envelope, an optional SOAP header and a mandatory SOAP body, as shown in Figure 2.9:

- *SOAP Envelope* is the top element of the XML document that represents the SOAP message. XML namespaces are used to disambiguate SOAP identifiers from application specific identifiers.

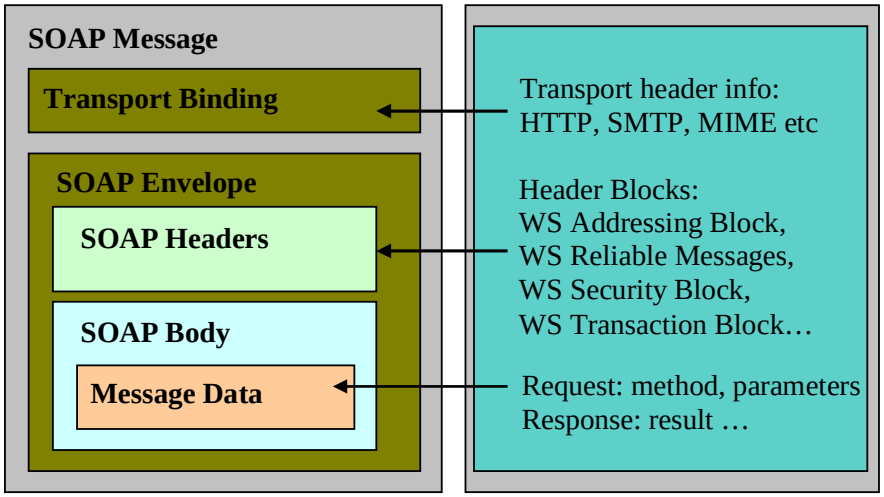

Figure 2.9: Structure of SOAP Message

- *SOAP Header* is an optional element that contains application specific information, like reliability and security blocks etc. If the header element is present, it must be the first child element of the envelope element.

- *SOAP Body* contains the actual SOAP information intended for the ultimate endpoint of the message. The body are a couple of XML element constructed in the pattern of request/response to realize the RPC functionality.

- *SOAP Transport Binding* specifies a concrete internet protocol to exchange SOAP message over Internet. HTTP is recommended for its popular acceptance, and other protocols are accepted, such as SMTP, MIME etc.

2.3 Workflow Management

Workflow management systems are software systems that support the definition, creation and execution of workflows in an organization. According to the Workflow Management Coalition (WfMC) [93], a non-profit international organization of workflow vendors, analysts and research group, workflow is defined as "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." Workflow normally comprises a number of logical steps, each of which is known as an activity that involve interaction with a user or workflow participant. The first idea of workflow originates to speed the time and improve the efficiency of office task processing. As the development workflow has evolved into different applications, such as production workflow to manage large numbers of similar tasks and to optimize productivity, administrative workflow to easily and flexibly define the process, collaborative workflow focusing on teams working together towards common goals and ad-hoc workflow to quickly and easily create and modify process definitions

as necessary. Nowadays workflow is also employed into services oriented computing to enable the integration and collaboration of business services [88].

2.3.1 Workflow Reference Model

Through years of work, WfMC reaches a common appreciation on the definition of workflow and develops a generalized target architecture driving the development of most production workflow solution. The reference model [42] defines a generic workflow application structure by identifying the interface within this structure to provide a standard for interoperability among the major workflow subsystems. Figure 2.10 illustrates the architecture of workflow reference model that contains the following major components and interfaces.

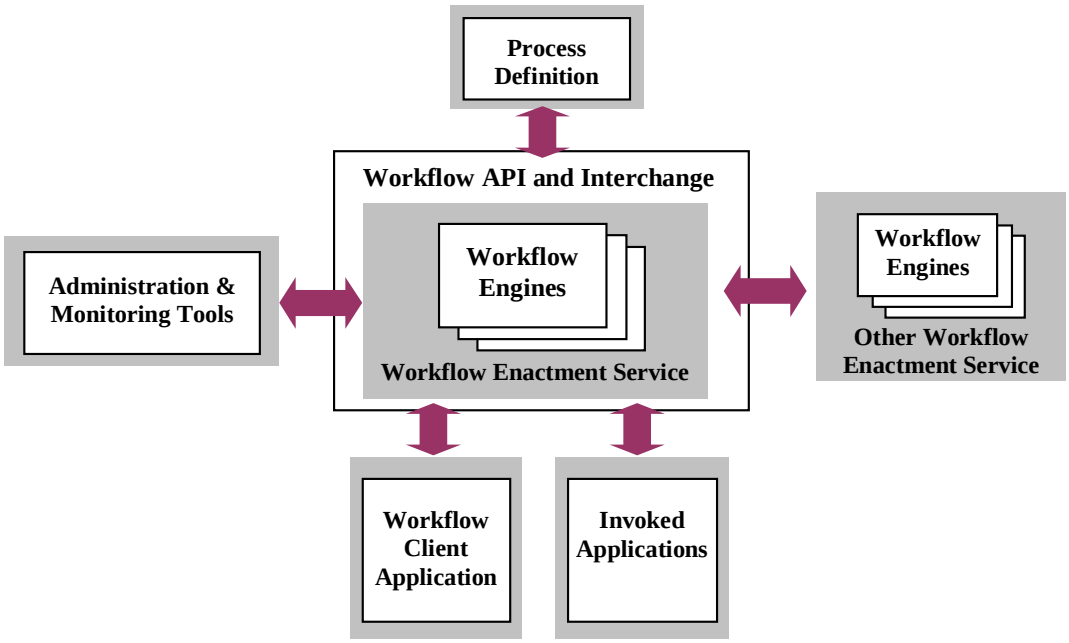

Figure 2.10: Workflow Reference Model - Components & Interfaces

Workflow Enactment Services

The workflow enactment service provides the run-time environment in which process instantiation and activation occurs, utilizing one or more workflow management engines and responsible for interpreting and activating the process definition. The primary component within enactment service is workflow engine which provides the run time execution environment for a workflow instance. A workflow enactment service comprises one or more compatible workflow engines with defined set of process definition attributes. The mechanisms of dispatching process execution are organized across the various workflow engines, protocols and specific interchange formats.

Process Definition

A variety of tools that may vary from the informal to sophisticated and highly formalized are used to analyze, model, describe and document a business process. To support the exchange of process definition information over a variety of interchange media, some interfaces concerning interchange formats are defined. These are process definition import/export interface which function between the modeling and definition tools and runtime workflow management software.

Administration and Monitoring

Reference model defines a common interface which enables several workflow services to share a range of common administration and system monitoring functions. These interfaces are intended to allow a complete view of the status of work flowing through the organization and present a comprehensive function set for administration purposes, including specific consideration of security, control and authorization.

Workflow Client Functions

In the reference model interaction occurs between the client application and the workflow engine through a well defined interface embracing the concept of a Work List - the queue of work items assigned to a particular user by the workflow engine. Activation of individual work items from the Work List may be under the control of the workflow client application or the end-user. A range of procedure is defined to enable new items to be added to or removed from the Work List.

Invoked Application

The concept of Application Agent is used for variety of method invocation behind a standard interface into the workflow enactment services. In this case application invocation is handled locally to a workflow engine, using information within the process definition to identify the nature of activity, the type of application and any data requirement. It is also possible to develop workflow enabled application to interact directly with a workflow engine.

2.3.2 Workflow Patterns

The application of workflow technology has evolved from business process modelling, coordination to component framework and business to business interaction. Workflow reference model proposed by WfMC presents a specification for workflow management system. However this reference model is only defined for its functionality aspect and other different control perspectives. The fundamental description and analysis of workflow are realized by a series of implemented workflow language, and the interpretation of basic constructs of workflow, such as sequence, iteration, split parallelism etc., is not uniform and clear. Even without formal qualification, the distinctive features of different workflow languages allude to fundamentally different semantics.

A practical and effective approach for defining and describing the existing workflow processes is workflow pattern. According to [80], a pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary context.

So far there is still no any standard to give a formal definition and categories for workflow pattern. Aalst et al. have done pioneering work for this [90], and define the workflow pattern in an abstract and uniform way that contains a description, synonyms, example, problem and potential implementation strategies [89]. They categorize most kinds of workflow patterns as follows:

- *Basic Control Flow Patterns* capture elementary aspects of workflow process control and closely match the definition of elementary control flow concepts in WfMC. Examples are Sequence, Parallel Split, Synchronization, Exclusive Choice and Simple Merge etc.

- *Advanced Branching and Synchronization Patterns* focus on advanced patterns for branching and synchronization which appears commonly in real life business scenarios. Examples are Multi-Choice, Synchronizing Merge, Multi-Merge and Discriminator etc.

- *Structural Patterns* illustrate typical restriction imposed on workflow specification and their consequences. Examples are Arbitrary Cycles and Implicit Termination etc.

- *Patterns involving Multiple Instances* reflect the workflow that an activity in a workflow graph can have more then one running, active instance at the same time. Examples are Multiple Instances without Synchronization, Multiple Instance with (without) A Priori Design Time Knowledge etc.

- *State-based Patterns* model workflow where instances are in a state awaiting processing rather than being processed. Examples are Deferred Choice, Interleaved Parallel Routing and Milestone etc.

- *Cancellation Patterns* uses a construct where one activity cancels another. Examples are Cancel Activity and Cancel Case etc.

Chapter 3

Evolution of Dynamism in Distributed Systems

Dynamism is an important feature of component software systems, which allows to flexibly change or reorganize the components of systems to meet the varying requirements of application environments. This chapter presents the evolution of dynamism in distributed systems from black box mechanism to configuration, dynamic reconfiguration and computational reflection. Also a more flexible and dynamic feature is presented: multi-solutions synchronously supporting that is realized in my novel approach of Routing Based Workflow.

3.1 Issues Introduction

To decrease the development cycle and alleviate the burden of distributed application developers from tedious non-business programming, middleware is proposed and widely applied to establish large scale distributed business applications. Middleware masks the common and complicated issues of distributed business applications, such as communication, security, synchronization and transaction etc. For distributed application programmers, middleware is taken as a black box - programmers just get what he wants from middleware APIs or services, and do not need to care about how these functionalities are implemented. The mechanism of black box had brought middleware widely utilized in enterprise applications. However, as emergence of new devices and development of new technologies, the demands and requirements of distributed applications get increasing and more complicated. Although middleware has got great success since it had been proposed, there appears a series of problems to spur the evolution of dynamism, such as:

1. How to make distributed systems modular and extensible? There are always new functionalities required to be integrated into the systems for the new requirements. The difficult for this integration is to avoid to make impacts to the legacy systems in terms of functionalities and source codes. Component oriented software architecture provides a solution direction to develop software

system with the capability of modularity, extensibility, easy to integration and management.

2. How to make distributed systems configurable and customizable for different application requirements? When designing and implementing a middleware system, developers always try to complete the functionalities and services as much as needed. But if all available components are loaded to run when the server starts, the fat server will cost much unnecessary resources for some seldom used components. A configurable system allows to assemble specified components to provide customized services for different requirements.

3. How to make distributed systems dynamically reconfigurable and adaptive to varying environments during run time? In some case, it is impracticable or will produce big loss if systems shut down or restart. But, components often have to be updated for a new version, or a new component is demanded to be integrated into a system. In this case, if the system has the capability of dynamic reconfiguration, the loss could be decreased to minimum.

3.2 Configurable Component Systems

Traditional middleware systems have fixed structure and comprise of fixed components to provide fixed services. This kind of middleware has good stability, but strongly restricts the range of its applications. Later configuration file is used to customize the components of middleware, and different component can be loaded to adapt different circumstance when middleware is started. However, the alternative component always shares the same abstract interfaces for functional behaviors with that of being replaced one [84]. For instance, there are several candidate communication protocol components, sharing common communication interface behaviors, provided in the configuration file. When the middleware system starts, only one protocol component will be chosen to load into the system to take charge of communication tasks. This mechanism of sharing the same functional abstract interface does not provide the real configurable ability because all the candidate components are associated in source level. So we call it half-configurable ability.

Architecture Description Language (ADL) [59] was proposed to support developing configurable distributed applications. An ADL is a language that provides features to model a software system's conceptual architecture which is distinguished from the system's implementation. ADL enables component based construction of large scale software system, and shifts the focus of developers from lines of code to coarse-grained components and system families. Explicit separation of components gives developers more flexibility for potential component reuse and substitutability. Some common referred ADLs are Darwin [57], Wright [76], Rapide [55], ACME [37] and so on. According to the classification framework for ADLs [59], any ADL should at least have three modeling elements: components, connectors, architecture configuration and a set of supporting tools.

- *Component* is modeled for a unit of computation and data store. It can be as small as a single procedure, or as large as an entire application. As an indispensable feature of ADL, component has to explicitly specify its interfaces which are a set of interaction points between the component and external environment. Some other optional feature may also be specified in a component: types of component, constraints and non-functional properties etc.

- *Connector* is modeled as a media to enable the interaction between different components and should also include some rules to govern these interactions. A set of interaction points have to be specified for connector's interface that enables the communication between connector and components. It's just the modeling of connector that keep the component real independent from others at level of source and thereby reason about architectural configurations.

- *Architecture Configuration* is modeled to construct the graph of components and connectors. It determines whether appropriate components are connected, their interfaces are matched, and connectors is enabled for proper communications etc.

- *Supported Tools* are, strictly speaking, not a part of ADL. However, the tools support architecture design, analysis, evolution, execution and so forth, so it directly affects the usability of ADL.

3.3 Dynamic Reconfigurable Systems

ADLs typically support only static architecture configuration and do not provide facilities to dynamically change the architecture. However, small part ADLs also try to provide a dynamic extension on their existing static specifications to widen the scopes of their applications. For example, C2-SADL proposed by Medvidovic et al [60] enables dynamic architecture changes by extending the ADL, C2 style, with an architecture construction notation (ACN). Darwin [52], [53] enables the feature of dynamic change by adding a set of meta-level Darwin specification on its original specification. Dynamic Wright [1] tries to solve the problem of dynamism by extending the Wright with some additional features. The dynamism provided by the extension of ADL is termed as programmed reconfiguration [26]. Because all ADLs have to be compiled into a executable system before they can run, the run-time changes to those architectures should be interpreted and can be predicted. In this case ADLs behave as a dynamic programming languages. The difference is that its changes occur at a level of granularity which is above of programming language statements. In the implementation, for instance, Darwin allows runtime changes of components via dynamic instantiation with different parameters, as well as via interpreting language scripts.

For more large scale distributed applications, the operations of dynamic reconfiguration are not unplanned and un-predicated. These change operations include, for example, updating components with new versions, removing the old component and

adding a new component. Some change operations can reach a further deep modification, such as updating the method behaviors of a component. Dynamic Reconfiguration can be said successful only if it satisfies the following requirements [66], [2]: 1) structure integrity to ensure the changed system in a valid state in term of component functional behavior; 2) consistency preserving to make sure the change processes are correctly carried out; 3) state transferring to guarantee the inner states of component are inherited from old one. So far, many approaches for dynamic reconfiguration have been presented, but there is still no solution to perfectly address above issues. We argue that the following aspects have to be considered as the important factors when designing an approach for dynamic reconfiguration:

- *Separation of Non-functional Concerns from Component* means the source codes for different concerns are not weaved tightly together which is important for the system to enable being dynamically changed. This separation permits the formulation of general structural rules for changes without the knowledge of application functional states. We have to notice that separation of different concerns of component does not mean the component has no codes for non-functional management. Non-functional concerns are just behaved independently.

- *Components Execution Model* describes how the components interact to each other. The execution model is greatly affected by the construction of components, and at the same time it also poses big impacts on the reconfiguration management. A more flexible execution model is possible to result in a more flexible reconfiguration management.

- *Reconfiguration Algorithm and Management* determines how the changes are dynamically made on the software system. In one side the reconfiguration management determines which kinds of operations are allowed to be carried out on the software system. This decides the flexibility of system to adapt the varying environment. In another side the reconfiguration algorithm decides how the change operations are carried out on the system quickly and with minimal disruption. Most work of reconfiguration algorithm focus on how to drive the reconfiguration involved system into a safe state in which change operations are able to be carried out.

Adaptive systems are based on a set of predefined configurations that have been assessed during development. During the period of execution, an adaptive software system is able to modify its own behaviors according to the predefined configuration in response to the changes of the running environments. Compared to the dynamic reconfiguration system, the adaptive system not only has to deal with dynamic changes management of the software system, and it is also responsible to plan and deploy the change conditions, monitor the system execution, and evaluate the environmental changes [73]. To be more specifically, an adaptive system has to realize the following functionalities:

1. *Planning Conditions for Changes*: the conditions for changes have to be predefined to let the system know in which case the changes should be automatically

carried out. In addition, the concrete change operations for different change cases have to be predefined during the system deployment.

2. *Monitoring the Execution*: adaptive systems need a monitoring entity to monitor the normal execution of system and also catch every change of the system and the running environment.

3. *Evaluating the Changes*: the evaluating entity closely collaborates with the monitoring entity and predefined conditions to make the decision when the environmental changes can be ignored and when the system has to be adapted for the changes.

3.4 Reflective Systems

Dynamic reconfiguration enables software systems to reconfigure its inner structure to deal with faults or meet the varying environment. However, in some case there is another demand to enable users looking up the inner structure of a system from outside environment, and then making the changes on the system based on the acquired structure information. This is the target of reflective system.

The concept of computational reflection, or simply reflection, was first introduced in the research field of programming language [56] to enable software systems to access, watch its computation and possibly alter its own interpretation to change the way it is performed. Currently the well known example is the Java package *java.lang.reflect* which supports only the introspection about the classes and objects in the current Java Virtual Machine [47]. With respect to the typology of actions performed by the meta-entities, the reflection can be classified into two branches: structural and behavioral reflection [19]. Structural reflection can be defined as the ability of a language to provide a complete reification of both the program currently executed, as well as a complete reification of its abstract data types. Behavioral reflection is defined as the ability of a language to provide a complete reification of its own semantics as well as a complete reification of the data it uses to execute the current program. In [10] Blair et al had done a pioneering work to introduce the reflection to middleware systems. In author's viewpoint a reflective system is one that provides a representation of its own behavior which is amenable to inspection and adaptation, and is causally connected to the underlying behavior it describes. Speaking more clearly, a reflective system holds the following two distinctive features:

Inspection Reflection can be used to inspect the internal behavior of a language or system. By exposing the underlying implementation, it becomes straightforward to insert additional behavior to monitor the implementation. This can be used, for example, to implement the functions of performance monitor and QoS monitor etc.

Adaptation Reflection can also be used to adapt the internal behavior of a language or system. This can either be achieved by changing the interpretation of an existing feature or by adding new features.

Since the conception of reflective middleware was first proposed, two frequently referred works are OpenORB - Reflection in Open Distributed System leaded by G. Blair from Lancaster University [10] and dynamicTAO Project from University of Illinois at Urbana-Champaign [50].

3.5 Multi-Solutions Supported Systems

Do the current approaches for dynamic reconfiguration and reflection address the problem of dynamic changes on software systems to adapt the varying environment? Although adaptive systems and reflective systems offer richer and more flexible capabilities, the core issue is still concentrated on dynamic changes. As introduced in the section 1.1, most existing approaches for dynamic reconfiguration adopt two kinds of mechanism to preserve the consistency during reconfiguration: 1) consistency through recovery; 2) consistency through avoidance. The first kind of mechanism allows to drive the system into a safe state immediately by broking the current interaction which will definitely result in certain loss. Nowadays most approaches abandon this kind mechanism for its obvious disadvantages, and turn to seek a better algorithm of second kind of mechanism. However, whatever the algorithms are proposed, the core idea of the second kind of mechanism follows a strategy of *waiting until safe state*, which includes the following steps:

1. Start to make reconfiguration and suspend the incoming requests;
2. Adopt an algorithm to drive all affected components into a safe state in which components finish their execution activities and are kept in a state of waiting;
3. Carry out change operations on the component software system;
4. Resume to execute the suspended requests by new reconfigured system;

The strategy of *waiting until safe state* works well in normal applications, but it may collapse in some extreme cases. For example, if an affected component is involved into a long time interaction, the processing of reconfiguration will have to last for long time, which perhaps leads to a terrible consequence.

In addition, considering a mobile application supported by an adaptive system, if an active user who is using the multimedia service goes into high bandwidth zone from low bandwidth zone, the adaptive system will immediately delete the component of buffer filter to provide higher quality multimedia service for the user. In this case, how to continuously provide the multimedia service for the users who still stayed in low bandwidth zone? In general, for a adaptive system, how to satisfy the old requirement for rest users after changes are dynamically made to meet the new requirement for partial users?

In this dissertation, I propose a novel approach named Routing Based Workflow (RBW) that is able to address above mentioned problems. Firstly, the RBW is able to make dynamically reconfiguration on component software system with minimal

expense: i) the reconfiguration processing can be finished in a predicated time; ii) the blackout time can always be limited in an extremely small time in any applications. Secondly, the RBW is able to support multiple solutions synchronously to provide personalized services for users.

Motivation

It is well proven in other area, such as IC, that the more independent a thing is, it can achieve more flexibility. Similar in developing software systems, if a component can be constructed more independently, it can get more flexibility in its execution and reconfiguration. To achieve more dynamism on software architecture, it not enough to separate the configuration concern from its functional implementation. It would be better to separate all non-functional concerns, such as configuration, execution management and even communication etc., from its functional implementation. In term of execution, I call these non-functional execution and management as execution environment of components. Every software component lives in its individual execution environment. When the execution environment of component is provided, the component is able to be executed there. For multiple collaborative components, I define a global execution environment that connects all individual execution environments and is able to transfer and exchange information among components. All collaborative components can run and interact with each other when their global execution environment is available. The execution environment could be duplicated, modified, replaced or even deleted, and these changes will naturally result in a new collaboration for all involved components. In my routing based workflow, the global execution environment is modeled as a routing. When a reconfiguration instruction comes, what need to do is to modify the routing and update the old routing with new one. The hard issues, such as consistence preserving etc., is then naturally simplified because all change operations are acted on routing, namely global execution environment, rather on components. The multi-solutions supporting can be realized by supplying multiple routings synchronously.

In the research field of programming language, a technology of dynamic binding or late binding is used to load the complied library at the running time [34]. The dynamic binding offers high flexibility, such as polymorphism etc., for programming language because the implementation of an interface can be determined during the execution of application. To further enhance the execution flexibility of component software, I introduce a concept of temporary binding for component instances which means the component instances are temporarily loaded into a created execution environment to execute their functions. After executions, the component instances will be released and returned to their repository to wait next executions. The temporary binding increases the independence of components, and also provides the higher possibility to enable dynamic reconfiguration with minimal expense.

Chapter 4

Routing Based Workflow

Routing Based Workflow (RBW) is a new proposed approach to enable dynamic reconfiguration on component software systems and support multi-solutions for components collaboration which is able to offer personalized services for users. RBW is the fundamental technology of our secure middleware system, and the system for dynamic services composition, which are introduced in next chapters. This chapter presents the RBW covering its structure, execution, management, dynamic capabilities, and extension for distributed components.

4.1 Overview of Routing Based Workflow

Routing Based Workflow (RBW) presents a novel software modeling approach for collaborative software components, and demonstrates how to manage and guide the execution of components within its modeling structure. The central advantage of RBW is to enable temporary binding for component instances, which results in high flexibility of dynamic changes on the system structure and simplifies lots of hard issues arising from traditional approaches for dynamic changes.

4.1.1 Definition of Routing

As it is well known, routing is a concept commonly appeared in Internet protocols where routing is an action of moving information across Internet from a source to a destination. For collaborative components, intermediate data are also required to be transferred from one component to another component deployed in the same computer or across networks. Here the technology of RBW is proposed to manage the execution, communication and control activities among different collaborative components, and the concept of routing is borrowed to describe the structure, dataflow and control dependencies among these components. From a given routing, designer or programmer can acquire information to know the organizational relation of one component with others. In a routing, each component is scheduled to execute its function in arranged orders and the intermediate data can be transferred according to the scheduled directions. From the viewpoint of applications of components, routing

can also be imagined as pipeline of intermediate data. After a routing is created, data can flow from one component to other components via pipeline of data stream. From the viewpoint of designers, a routing models a global execution environment for all collaborative software components. In a routing, there is no persistent component instance. Component instance is only temporarily loaded into a routing, namely a global execution environment, when necessary for execution, and will be unloaded from the routing immediately after execution. The execution of a routing brings sequent executions of series of components in a scheduled order. At anytime when a request comes, a routing is assigned to execute this request. If an instruction comes to change the collaboration relation of components or replace one component with its new version, what needs to change is only the routing, i.e. the global execution environment, instead of component instances themselves.

4.1.2 Framework of Routing Based Workflow

The core element of RBW is routing which models the structure, execution, communication and control of collaborative components. In order to construct a routing, the workflow management technology is employed to manage the organizational relation of different components. In addition, software patterns such as proxy pattern and pattern of object pool etc. have been used to construct components in routings. Several aid modules are also created to manage and guide the execution of components via processing of routings. Figure 4.1 describes the framework of RBW and shows how different modules of RBW work together. Before explaining the working mechanism of RBW, some terminologies are clarified as follows:

- *Workflow Manager* takes care of overall work of RBW. However, it is not involved in specific functions realized by other modules, and concerns more on the organization of other modules. Concretely speaking, Workflow Manager is responsible to initialize the RBW to accept requests, organize relevant modules to execute requests, and send back responses.

- *Routing Schema* describes the structure of routing and the constituted elements. It exists in two forms in RBW. One form, called XML-oriented routing schema, is described in offline configuration file using XML based configuration language, see section 4.5 for details. The other form, called language-oriented routing schema, is described in programming language, such as Java in our case. These two forms are kept consistency. While RBW is initialized, offline configuration will be parsed into language-oriented routing schemas. Similarly, when server is going to shut down, language-oriented routing schemas may be changed and then they have to be rewritten into XML-oriented routing schemas stored in configuration file.

- *Schema Lib* serves as a schema repository that stores only language-oriented routing schemas. It provides a set of interfaces to conveniently insert, get or modify routing schemas. Its collaborated module is schema configurator.

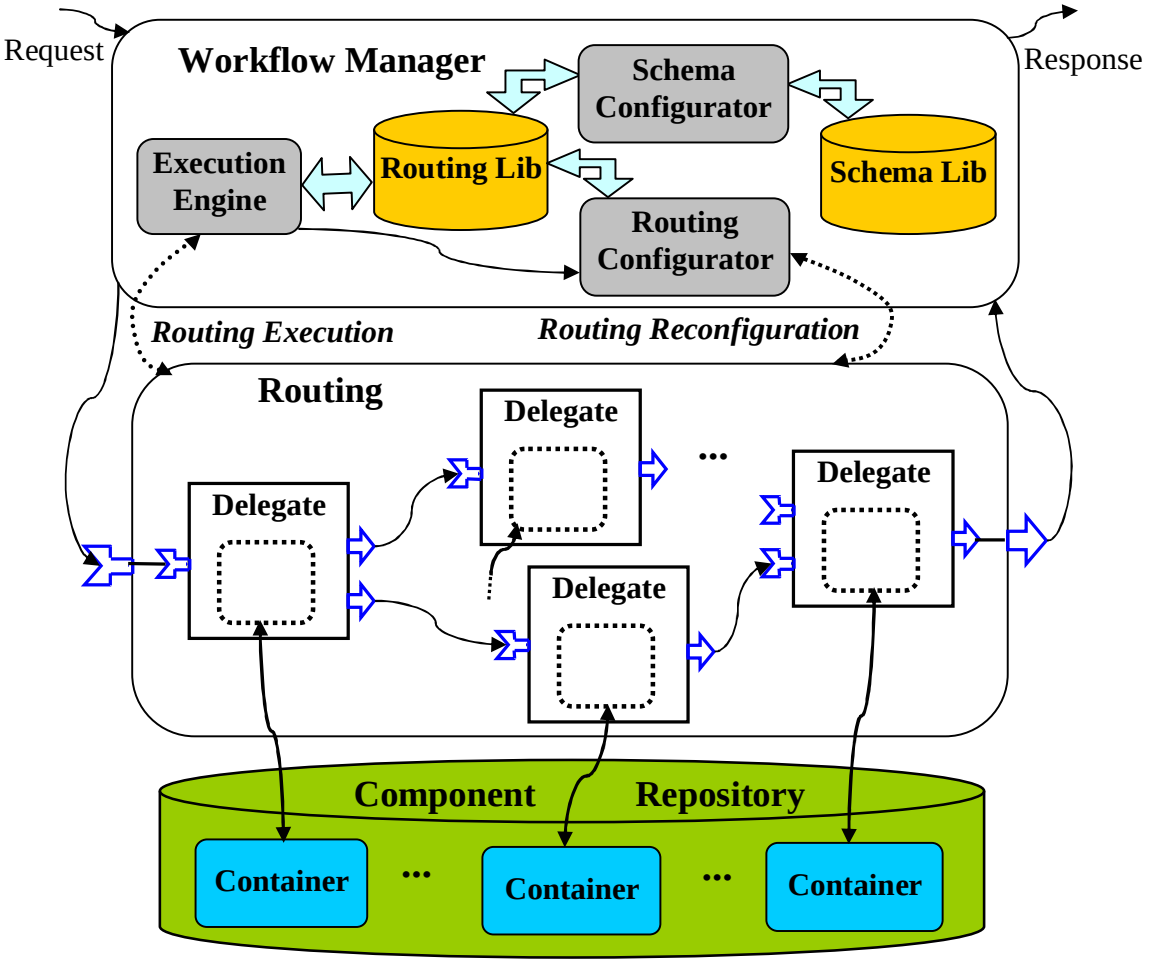

Figure 4.1: Framework of Routing Based Workflow

- *Bound Routing* models the global execution environment for all involved components. In a bound routing, each involved component has a representative to establish the communication path among components. All component instances are just temporary loaded into a bound routing to execute their functions, and then unloaded. The round routing is responsible for all non-functional tasks, such as data and control message transferring etc., to guide interaction and management among component instances.

- *Routing Lib* is similar to schema lib, but what it contains is bound routings instead of routing schemas. The routing lib collaborates with execution engine which fetches specified bound routing from lib for every request, schema configurator which stores a bound routing to lib after it instantiates a routing schema, and routing configurator which acquires a copy of a specified routing from lib for change and updates the routing after changing.

- *Schema Configurator* provides more functions than what its name hints. Firstly, schema configurator takes charge of parsing functions to parse routing schema

from XML based configuration file and coding functions to rewrite routing schema to XML based configuration file. Secondly, schema configurator is responsible to instantiate a routing schema into a bound routing. Thirdly, schema configurator fulfills the reconfiguration operations on routing schemas according to the changes made on the corresponding bound routings.

- *Routing Configurator* implements the dynamic change operations on bound routings. These change operations concern adding, deleting or replacing elements of a bound routing. The changed routing should also be a valid and workable routing that will result in a new collaboration relation for all involved components. The change interface provided by routing configurator could be directly used by Graphic User Interface (GUI) based reconfiguration tools, or be accessed from authorized client requests via execution engine.

- *Execution Engine* drives the execution of all involved components via processing on a routing. Inside of execution engine, each routing has a pool of routing processor to deal with the real processing on a routing. When a request comes, a specific routing processor will be assigned and initialized with a request. The thread technology based routing processor is then started in a parallel manner for processing: triggering the execution of components by the data arrival from request or other executed components.

- *Component Repository* is a virtual repository for all available components. Actually, all components are hosted in their own container independently. Components could be deployed in the same computer or in different computers.

Before RBW starts to work, it has to be initialized. The step for initialization can be divided into two parts: parsing from offline configuration and routing instantiation. After parsing from XML based offline configuration, routing schemas are re-expressed in a format of programming language. The process of instantiation makes the routing to be a workable object - bound routing. The work mechanism of RBW is simply illustrated in Figure 4.1, where block arrows indicate the interaction between processing modules and lib modules, black lines with directed arrow indicate the data flow direction. Two dashed lines are also used to indicate an indirect operation concerning routing. For example, execution engine executes routing via a third module - routing processor, and routing configurator makes changes on a copy of the specified routing. In RBW, a request contains routing ID that indicates which routing will be used to execute for this request, and input parameters that will be consumed during the execution of this request. When a request comes, the workflow manager firstly acquire a bound routing from routing lib according to the acquired routing ID, and then assign a routing processor to execute this routing with input parameters contained in request. The workflow manager does not need to wait for the result of execution because routing processor is designed using thread technology and knows how to send out response with the information provided in request. When another request comes immediately, workflow manager can accept it soon and assign another routing processor with a corresponding bound routing to deal with

it. The key technology of RBW is the structure and execution mechanism which are introduced in subsequent sections.

4.1.3 Three-Layers Modeling

RBW models the process of collaborative components from their static structure to running time states, as shown in Figure 4.2. The modeling of components in different stages can be reflected into three tiers: routing schema, bound routing and active routing. Routing schema describes the static structure relation of collaborative components, and specifies features and properties of each component. Bound routing is a routing in which all components are instantiated, and interfaces of all components and their data communication path are established and tested. In other words, in a bound routing all components are waiting for upcoming requests and ready to run. The third tier, active routing, is the execution environment where the execution, communication and control activities among all involved components occur. Workflow manager is more like a coordinator to coordinate different modules to execute their tasks in different layers. Comparing the second layer - bound routing and the third layer - active routing, we get more clear that a bound routing is just an idle global execution environment where there are no any ongoing execution activities of components. That means RBW does not need to wait any more and can immediately get a *safe state* software execution environment. The change operations that realize dynamic reconfiguration functionalities performed by routing configurator can be immediately acted on a bound routing, an idle execution environment, instead of components themselves. So a series of hard issues arising from dynamic changes, such as component states transferring and consistency preserving etc., can be avoided or greatly simplified in RBW.

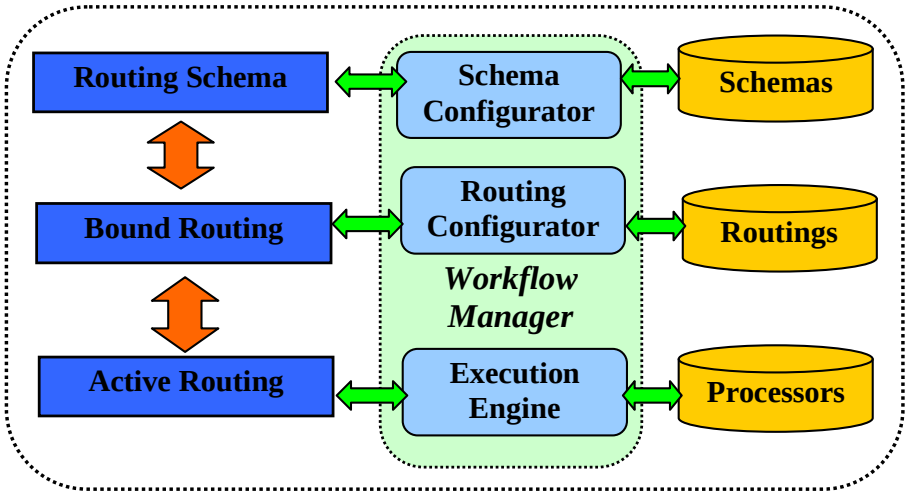

Figure 4.2: Three Layers Modeling of Routing Based Workflow

4.2 Routing Structure and Modeling

In this section I describe the constituted elements of routing, and show how routing can be used to model the collaborative software components. The routing structure and modeling discussed in this section and next three sections concern only the components deployed in the local computer. This is also the application case of RBW in our secure middleware construction introduced in chapter 5, and dynamic web services composition introduced in chapter 6. In section 4.6 an extension of RBW will be introduced for components deployed separately.

4.2.1 Routing Structure

The construction of routings adopts the graph oriented workflow structure. In RBW, the managed components can be randomly integrated together if the IO interfaces of components are matched. As depicted in Figure **??**, all components are not integrated directly together. They use their representatives, component delegates, to integrate as a routing. The component instance is wrapped into component processor and hosted in its home, component container. The interactions among different components are guided by the routing. Actually routing works as a inner manager and takes care of the direct management activities on components. In the following subsections I introduce in detail each constituted elements of routing: component delegate, component container and component processor.

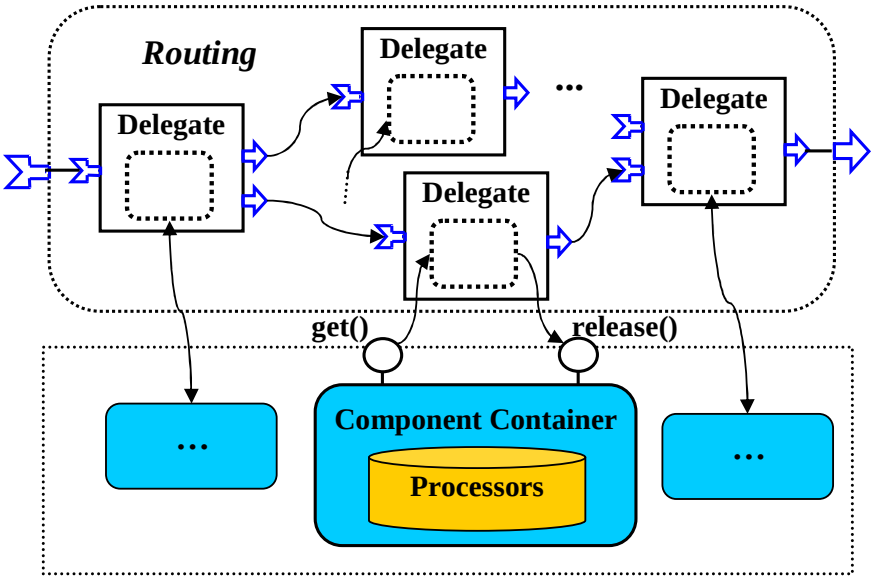

Figure 4.3: Structure of Routing

Component Delegate

Component delegate represents a component in a routing for all management activities. In a routing, what you can see is only component delegate and there is no

persistent component instance. As representatives of components, component delegates establish the connection among different components and accepts the input data from request or outcome of other components. The design and implementation of component delegate are greatly affected by the structural design pattern of proxy.

Proxy Pattern The proxy design pattern [14] makes the clients of a component communicating to a representative rather than to the component itself. The representative offers the interface of the component but performs additional pre- and post-process, such as the access control checking etc. Figure 4.4 depicts the general model of proxy pattern. The client is responsible to invoke a specific task. To complete this job, it invokes the functionality of the original component by accessing the proxy. The proxy offers the same interface as the original component, and maintains a reference to ensure correct accessing. The abstract original means the interface implemented both by the proxy and the original component.

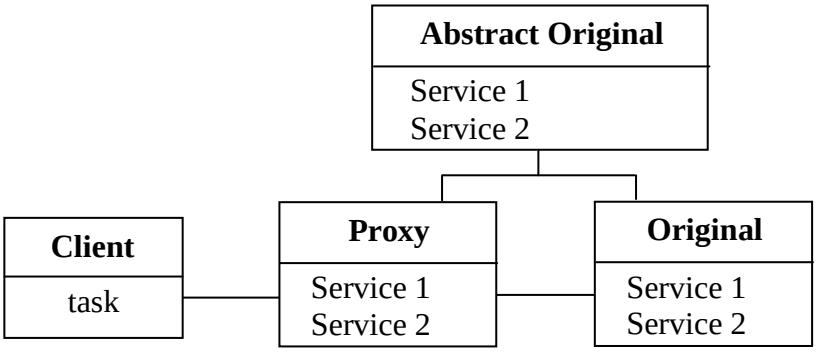

Figure 4.4: Diagram of General Proxy Pattern

Depending on the context in which proxy object is used, the proxy pattern can be divided into several different types, but commonly used are the following three types:

- *Remote Proxy* provides a reference to an object located in a different address space on the same or different machine. For example, a proxy for web services invocation shields the client from the remote address of web service and SOAP communication for message transfer.

- *Virtual Proxy* allows the creation of a memory intensive object on demand. The object will not be created until it is really needed.

- *Protection Proxy* provides different clients with different levels of access permission to a target object.

Design of Component Delegate The design of component delegate concentrates mainly on two aspects: service interface keeping and reference maintaining.

The classical approach to keep the service interface of proxy with that of original component is to share the same interface. Speaking in Java programming language,

the class of proxy and class of original component have to implement the same interface. However, there is no distinct invocation by reference in RBW. All the interactions among different components are realized by the data driven communication and execution, which means component delegate only forwards the data from other component to its original component instance and the data arrival automatically trigger the execution of component. The design of keeping the same interface of component delegate with its original is realized by sharing the same communication ports which include in port and out port, refer details in next two section. A series of in ports and out ports are separately managed in two customizable arrays inside of component delegate. The class of component delegate is unique, but it can adapt to countless original components by changing its IO behaviors. When a component delegate is instantiated, a set of mandatory parameters have to be given to indicate which original component is delegated and which ports are used to represent the IO behaviors of this component. It is possible to create multiple instances of a component delegate to represent the same original component. In this case the original component may serve for multiple delegates during different time slices. This multiple delegates for one original component may happen when the component is used in different position of a routing or in different routings.

Another responsibility of component delegate is to maintain the reference of the original component through which inputting data and produced results can be forwarded. In component delegates the design of reference maintaining is also different from traditional methods. Component delegate does hold a reference. However, this reference does not point directly to the original component, but to a component container, detailed in next section, that manage the life cycle of the original component. Such design greatly increases the independence of component and also brings much flexibility for dynamic changes and reorganization of components. Through the reference to a component container, the corresponding component delegate can load the original component to execute its functionality when necessary. After execution the component delegate will immediately release the original component through this reference.

As the representative of component in a routing, component delegate also takes charge of other tasks to guide the execution of components and manage the control activities: for example, real binding and unbinding for component instance and control messages transferring etc., refer section 4.3 for details.

Component Processor and Container

Component processor is the wrapper for functional component entity that implements the functionality it claims. Functional component contains only the implementation of pure business logics, but it need to be wrapped into component processor to enable contact and communication with other components in a routing. The functions of component wrapper can be summarized into four aspects:

1. creating IO interface via communication ports;

2. checking execution requirements for component;

3. triggering the execution of functional component when all required data arrive;

4. pushing the results to next components after execution.

The closely collaborated module with component processor is its host, component container, which holds a pool of component processor instances. Component container takes charge of the life cycle management for component processor:

1. providing interface for client, here namely component delegate, to get from and release to pool;

2. enabling creating or deleting instance of component processor when necessary;

3. managing the size of pool and synchronization etc.

The design of component container is greatly affected by software design pattern of object pool.

Pattern of Object Pool The design pattern of object pool [36] provides an effective means to improve the execution performance by supporting the reuse of multiple instances of objects. Object pool is used in programming environment where the specific type of object is expensive to instantiate or only a limited number of this kind of objects can be created. Figure 4.5 shows a general diagram of the design pattern of object pool, where the client is any object in the system. To request a object, a client just calls *get()* method on pool manager and gives the object back to the pool by calling release method on the pool manager. The pool manager has to be initialized by a series of parameters that specify the class of the pooled object and relevant parameters for the instantiation of pooled object.

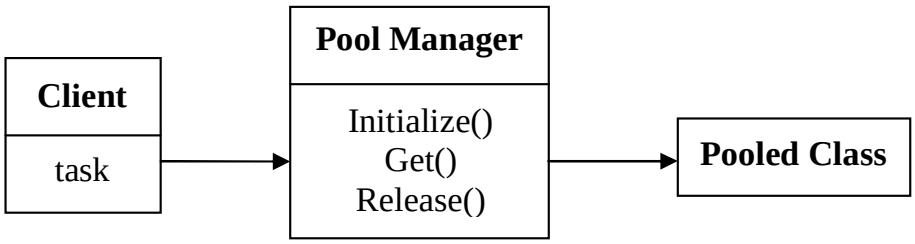

Figure 4.5: Diagram of Pattern of Object Pool

The pattern of object pool is most often used in two cases: network connections and database connections, which both cost much time and resources for new instantiation and have limitation for the number of connections. One of the well known applications of object pool pattern is the EJB container of J2EE which is a runtime container for deployed EJBs. During the entire life cycle of an EJB object, from its creation to removal, it lives in the container. The EJB container also provides a standard set of services, such as caching, concurrency, persistence, security, transaction etc., for EJB object used in enterprise applications.

Design of Component Processor As illustrated in Figure 4.6, component processor comprises of a functional component implementing pure functional business logics, and a generic processor implementing non-functional aided tasks for execution management. The component processor's IO behaviors with other components are realized through its in/out ports. Few methods of interface of component processor are offered for external objects to invoke the methods of component. Most other methods are internal services for guiding the execution of functional component.

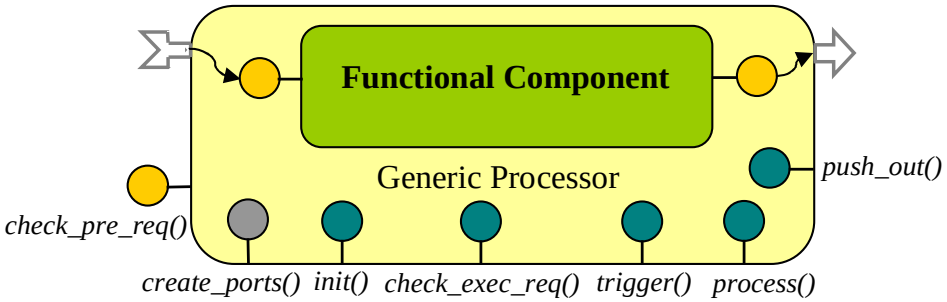

Figure 4.6: Component Processor

Generic processor is designed as an abstract class that has to be extended by all functional components to create function-specific component processors. The tasks of generic processor can be classified into two types: basic inner methods that are same for all functional components, and abstract inner methods that have to be implemented in different functional components. The basic inner methods of generic processor are listed as follows:

- *Offering high level communication ports creation API*: Given the information from IO interfaces of functional component, the high level port creation APIs are provided to create in ports connecting its suppliers and out ports connecting its consumers. During the procedure of ports creation, a series of inner in/out objects are also created in generic processor to accept the source data from in port and then forward to functional component, or accept the result from functional component and then forward to out ports.

- *Non-functional initialization*: The non-functional initialization means the data for initialization is beyond the functional execution of component and it is for execution preparation, such as the clearness of in/out objects created to connect IO ports, state initialization for component etc.

- *Triggering the execution of functional component* is independent from functional component itself. Every time the input data arrives, the in port will start triggering method to check whether all required data are arrived or not. If the answer is yes, the execution of functional component is triggered. Otherwise, just keep waiting for next data arrival.

- *Pushing out result* is the last step for the execution of component. The result will be set to the out objects bound to out ports, and then the out ports will

push the result to its next bound in ports which belong to other component delegates or response.

The abstract inner methods are mainly about function-specific behaviors of component and have to be implemented in each concrete component. The implementation of abstract methods also indicates the key steps to create a new function-specific component processor for RBW:

- *Creating IO ports*: component processor uses the high level ports creation API to customize its function-specific IO behaviors. The implementation for IO ports customization is realized in the constructor method of a concrete component processor with provided information of method parameters, such as name and type of IO objects etc.
- *Functional initialization*: the properties of a concrete component will be used to initialize a instantiated component processor.
- *Checking the preparation requirements*: this is provided for the component that needs specified condition or resources to enable the execution of component: such as CPU or memory requirements, sound cards or other special resources etc. This step is invoked by component container when carrying out the virtual binding for routing, as shown in section 4.3.
- *Checking the functional execution requirements* according to the acquired data from other components or request. This step happens during the period of execution.
- *Execution process* starts the execution of functional component with received data from its in ports and puts the result to its out ports that are connected to in ports of next component delegate.
- *Implementing the release function* to clear all produced intermediate data and to reset all relevant data as initial sate.

Design of Component Container Component container is the host and the only directly collaborated module of component processor. The temporary binding between component delegate and the instance of component processor, also called component instance, is created and controlled by its container whose reference are kept in the delegate. The component container maintains a pool for instances of one function-specific component, and each component processor has its exclusive container. The design of component container adopts the idea of object pool pattern, and comprises of three key functions as follows:

- *Creating component instance*: the fixed number of component instances have to be created in the beginning of instantiation of component container. During the runtime, there may also be a demand to create a new instance when there is no idle instantiated component. To enable component instance creation, a set

of relevant information have to be specified after creating an empty container. Such information includes not only the name and class of component object, but also some parameter objects that are required for component instantiation.

- *Maintaining a pool for instances*: two external methods, *get()* and *release()*, are the key to maintain the pool of component instances. The concrete tasks include initializing the pool, synchronizing operations of *get()* and *release()*, and maintaining two lists: locked list for dispatched components and idle list for available components.

- *Checking the compatibility with corresponding Component Delegate*: this compatibility checking is specially designed for RBW in which virtual binding is an important operation to make different component delegates connected. Although after virtual binding the component instances are still separated from routing, component container will make a compatibility checking to ensure that every component instances are bindable and matched to corresponding delegates. The implementation of checking is realized by temporarily loading a component instance to make real binding and unbinding. During the checking, an external method of component processor, *check_prepare_requirement()*, will be invoked to check whether the current host environment meets the execution requirements of component.

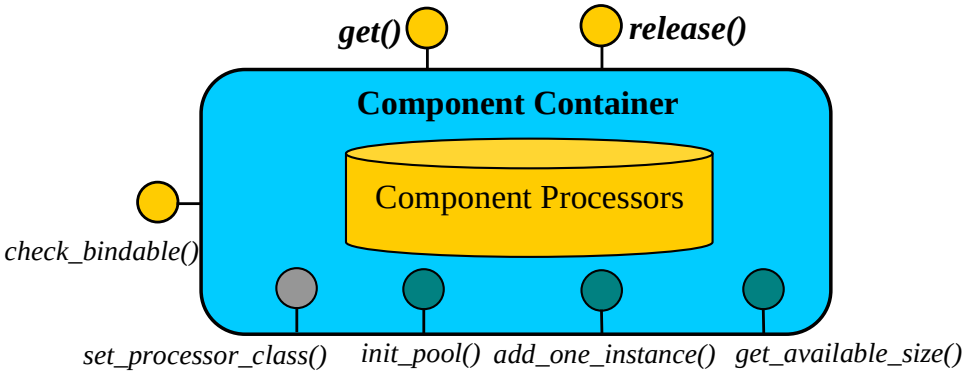

Figure 4.7: Component Container

Communication in Routing

In RBW, all the intermediate data flow from one component processor to others through their corresponding component delegates. The communication among different component delegates and the communication between component processor and component delegate are realized by communication ports. All the intermediate data transferred in routing are packed with a universal wrapper - Named Object.

Communication Port According to the data transferring direction, communication ports can be classified into in ports and out ports. Communication ports are used both in component delegate and component processor. For component delegate, the in ports is defined as port that accepts data from other component delegates and forwards these data to its represented component processor. For component processor, the in port is defined as port that accepts data from its represented component delegate and forwards the data to its inner functional component. The definition of out port is similar but the transfer direction is contrary, as shown in Figure 4.8. According to the type of transferred data, communication ports can also be classified into operation port and stream port. Operation port is defined as port in which the transferred data is an object that will be forwarded immediately after it is received. However, in the stream port the transferred data is stream data that will be continuously transferred and the forwarding and receiving operations are performed concurrently. So far, all of my work focuses only on the data transferring by operation port. In the rest of dissertation, if a communication port is referred, it is automatically considered as an operation port.

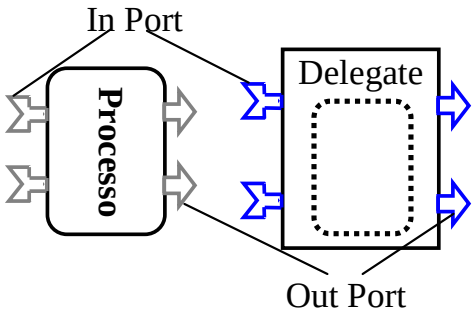

Figure 4.8: Communication In Port and Out Port

The working mechanism of communication port is described from its structure, interfaces and management as follows:

- *Port Structure:* Member variants of port are described in Figure 4.9, which is the Java expressed abstract class. Two key variants are *m_sourceList* that contains a list of source objects from which data are sent, and *m_destinationList* that contains a list of destination objects to which data will be forwarded. The source objects and destination object are mainly communication ports or the inner objects of component processor that are bound with ports and interacted directly with the functional component. In a port the transferred data will not be stored, and only pass through from its source to a series of destinations. Another three important member variants of port are: *m_owner* indicates to which this port is belong, *m_type* indicates what type of data can be transferred by this port and *m_state* indicates different states in which port stays during the execution of routing.

- *Port Type:* For each communication port, a port type is specified when a port is created. The information of port type contains: the IO direction, symbol

```
public abstract class OperationPort{

    private InBehavior m_owner = null;
    private LinkedList m_sourceList = null;
    private LinkedList m_destinationList = null;
    private PortType   m_type = null;
    private PortState  m_state = null;
    private Property   m_property = null;
    ...
}
```

Figure 4.9: Members of Abstract Class of Operation Port

indicates whether it is an operation port or steam port, and the data type of transferred object. When a port is created, it can only be bound to another port type matched port, and transfer the specified type of data. Otherwise, an execution exception will be thrown out. The port type is designed to ease managing the ports and to enhance the validation condition of ports binding.

- *Port Interfaces:* The interface of operation port is listed in Figure 4.10. Two important methods are *bind_source()* which is used to check and bind the source port together, and *bind_destination()* which is used to bind the destination port together. The method of *bind_owner_source()* is designed specially for out port of component processor to bind its source with its inner object consumed directly by functional component. Likewise, the method of *bind_owner_destination()* is designed for in ports of component processor to bind the result produced by functional component with port's destination. Another two important methods are *unbind_source()* and *unbind_destination()* that separate the ports of component processor from the ports of component delegate and eventually unload the component instance from routing.

```
public interface OperationPort{

    public int bind_source(OperationPort port);
    public int unbind_source(Object source);
    public int bind_destination(OperationPort port);
    public int unbind_destination(Object destination);
    public int bind_owner_source(NamedObject source);
    public int bind_owner_destination(NamedObject destination);
    ...
}
```

Figure 4.10: Interface of Operation Port

- *Port State:* During the execution of components, series of activities are acted on ports: ports binding, data transferring and ports unbinding. To efficiently manage these activities, the port state is designed to record the state and change state according to the execution stages of relevant activities. There are five states which are designed as follows:

1. STATE_INITIAL: is the initial state after port is created.
2. STATE_SOURCE_BOUND: is the state after successfully carrying out method of *bind_source()* or *bind_owner_source()*. In this state port is not allowed to transfer data.
3. STATE_DESTINATION_BOUND: is the state after successful carrying out method of *bind_destination()* or *bind_owner_destination()*.
4. STATE_READY: After source and destination are bound, port goes into this state. In this state, port is ready to forward data or to be unbound from other ports.
5. STATE_ACTIVE: when a port receives data, it goes into active state. After forwarding, it returns back to ready state. In active state, the port is not allowed to perform port unbinding operations.

- *Ports Management:* Ports are used both in component processor and their delegates. To improve the efficiency of implementation, a class of IOBehaviorAbstract is designed to take charge of the management task of ports and will be extended by all objects that facilitate the IO behaviors via ports, such as component processor and component delegate. The task of ports management contains common basic operations on ports: offering low level API for port creation, maintaining a set of list for different type of ports, enabling searching functions for available ports etc.

Data Transfer Wrapper - Named Object In RBW, all the intermediate data are not nakedly transferred via ports, they have to be wrapped into a Named Object and then be transferred. The Named Object helps to manage the transfer process. Speaking more clearly, what is transferred among the ports is an instance of Named Object which carries the data value and other relevant information. Named Object could also be constructed as the inputs set and output set of a routing, which would be bound to the ports of component delegates, and the inner objects of component processor that connect the ports of processor and consumed by functional component. The key constituted elements of Named Object are listed as follows:

- *Identifier* is the name of transferred object, and it is unique and represents object in RBW.
- *Type* indicates the data type of transferred object which will be used to check the validation of data transferring.
- *Value* stores the real value of the object.
- *State* has only possible two values: INITIAL and READY. Only when the state shows READY which means data are already set, the instance of *Named Object* can then be transferred via ports.

Control Links

In RBW, communication ports provide powerful functionality to create data flow pipeline and enable the interaction among different components. However, in some case the flow path can not be decided until it gets the execution result of last component. For example, only after the authentication is true, the data will flow to credit card payment component. Otherwise data will only flow to exception handling component. Here the control link enables flow selection by making simple logic computation with inputs and sending out the results to one of different ports which are connected to different components. The implementation of control link is realized using a similar way as component processor. But the control link has no delegates and container. Ports of control link are directly connected to ports of component delegates. In fact, control links can also be, in some extent, taken as simple and commonly used components.

Three pick links, i.e. AND Link, OR Link and XOR Link, and one data type mapping link, i.e. Map Link, are introduced in next paragraphs. All three pick links accept only boolean type of data and produce boolean type of results. For all three select links, the number of out ports are fixed with two and the produced values are also fixed: the first port only exports boolean type value of TRUE and the second port only exports boolean type value of FALSE. When pick link finishes its logic computation, it will only export one value: True or False. The out port acquired a boolean value will forward the value to its connected component delegate and trigger the execution of the picked component. The component that is connected to another out port of control link will then be blocked because there is no data sending out from another out port.

AND Link AND Link realizes the same function as logic operation: AND, as shown in Figure 4.11. AND Link accepts Boolean values from in ports, and outputs Boolean value 'TRUE' only if all the inputs are 'TRUE'. Otherwise, the output would be 'FALSE'. In AND Link the out ports are fixed, and the in ports can be freely customized with the restraint of fixed data type: Boolean. AND Link can have any number of in ports.

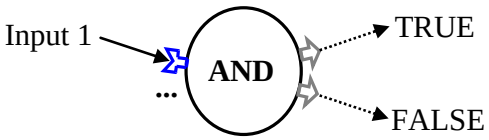

Figure 4.11: Diagram of AND Link

OR Link Similar to AND Link, OR Link realizes the function as logic operation: OR, as shown in Figure 4.12. OR Link outputs 'TRUE' if at least one input is 'TRUE'. The number of in ports could be any digit bigger than one.

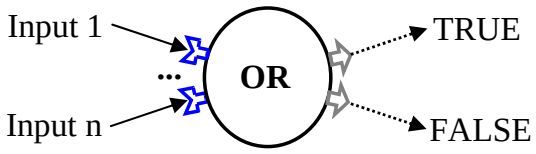

Figure 4.12: Diagram of OR Link

XOR Link XOR Link realizes the function same as another logic operation: XOR, as shown in Figure 4.13. If all inputs are same, the XOR Link outputs 'TRUE'. Otherwise, XOR Link outputs 'FALSE'. In XOR Links, the number of in ports must be two.

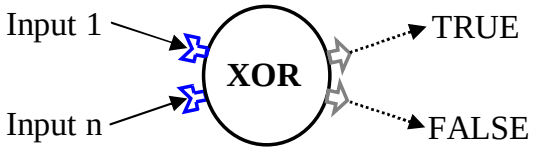

Figure 4.13: Diagram of XOR Link

Map Link For two collaborative components, the ports planned to be bound are possible to face a problem of type-mismatched. For example, one in port is for price expressed in string type, however, the potential coupled out port is expressed in integer type. In most cases it would be difficult to change the data type of any port. The Map Link is designed, as shown in Figure 4.14, to address such issues. The in port and out port can be customized freely for any simple data type. After customization, given a data with specific type, the Map Link can output the same data with expected type. The implemented data types in Map Link are nine simple data type: String, Short, Integer, Long, Float, Double, Boolean, Byte, Character. Notice that some data types mentioned above are not transformable, e.g. Double value can not be transformed to Boolean value.

Figure 4.14: Diagram of MAP Link

4.2.2 Routing Modeling

In RBW, components are represented by their delegates and the flow structure of collaborative components are modeled by routing. For the flexible construction of routing and specially designed communication ports, routing is a graph-oriented structure and is able to model most kinds of flow structure that reflects the process of components. Three typical flow structures modeled by routing are introduced in next

subsections: sequential and parallel routing, flow pick routing and cycled routing. The nested structure can also be modeled by routing in which a compound delegate comprises of several different component delegates. But this nested structure makes no much sense to improve the efficiency of execution and express capability of component process, so it will not be introduced.

Sequential and Parallel Routing

In the workflow management, sequential and parallel orders are two simplest and most commonly used structures for business process. Routing also naturally models these two basic flow structures. Sequential process is modeled in routing through data flowing from out port of one component delegate to in port of another component delegate. In a routing, two or more out ports of the same component delegate can be bound to in ports that belong to different delegates. Likewise, two in ports of the same component may also accept data from different components. In above cases, parallel process is modeled. An example of parallel process modeled by routing is depicted in Figure 4.15, where the directed arrows indicate the data flow direction.

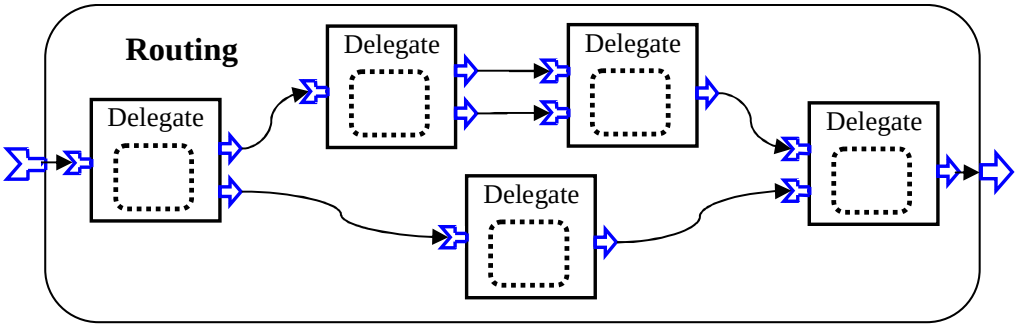

Figure 4.15: Example of Parallel Routing

Flow Pick Routing

Picking flow often appears in business process. The result of key component, such as component of authentication, will dramatically affect the process order of subsequent components. With control link, routing can rather perfectly model such picking flow. Three control links can be used to make decision according to the inputs and give a output direction to which data will flow. The other flow paths will be blocked because of waiting for the never arrived inputs. Notice that each component delegate will trigger the execution of component only after all of input data arrive. An example of flow pick routing is given in the following Figure 4.16, where the real directed arrows indicate the default data flow path and the dotted directed arrows indicate another candidate and possible flow path.

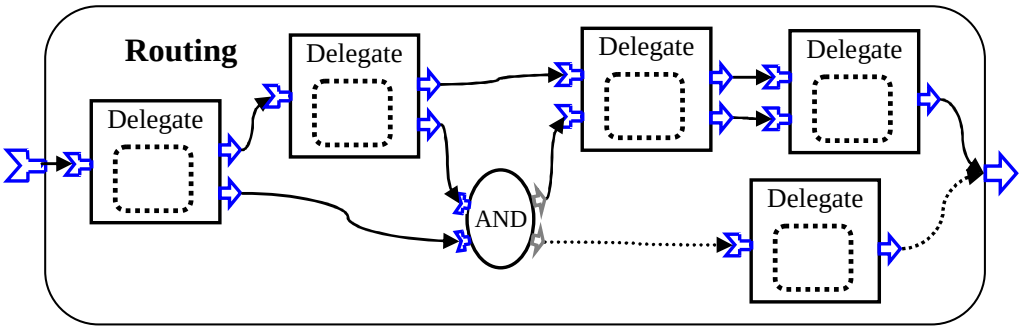

Figure 4.16: Example of Flow Picking Routing

Cycled Routing

In some cases, one or several components have to be executed repeatedly. Routing can model this kind of process by a cycled structure. The out ports of component delegate can be freely connected to any in ports if necessary and ports are matched. When the out ports link to in ports of the same component or the former component, the connected routing becomes a cycled structure. Normally, the implementation of component is a black box for the routing designer and the inner implementation is hardly used to make decision when the flow in the cycle will end. Just as in flow pick routing, in cycled routing control links once again play an important role to decide the time and condition to end a cycled flow. An example of cycled routing is given in the following Figure 4.17, where the real directed arrow indicate the cycled data flow path, and dotted directed arrows indicate the data flow path after the ending of cycle. Notice that, although *Delegate3* always sends data to one in port of *Delegate4*, *Delegate4* will not be triggered to execute during period of cycling because it is still waiting for another input from AND Link while cycling.

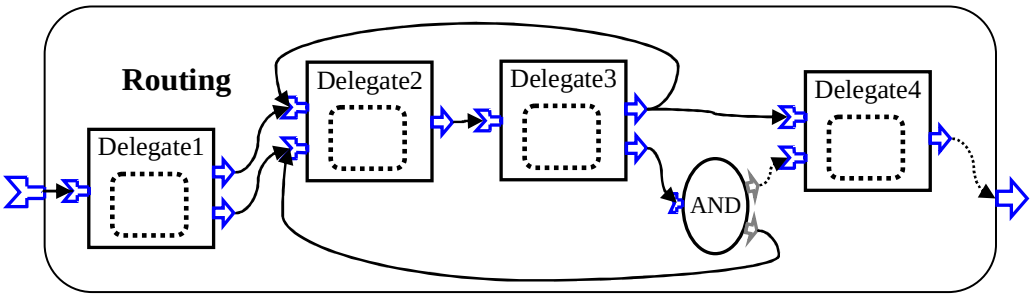

Figure 4.17: Example of Cycled Routing

4.3 Routing Execution and Management

Based on the routing structure and routing modeling introduced in the previous sections, the routing execution procedure and dependency management of routing are explained in this section. From explanation of routing execution, it is easy to

know how temporary binding of component instances are realized in RBW and result in dynamic reconfiguration with minimized prices.

4.3.1 Routing Execution - Temporary Binding

The execution of routing guides the executions of a set of collaborative component instances. Routing has three states during the period of execution: initial state, bound state and active state, which indicate different stages of component execution. Once routing is created and instantiated from routing schema, it goes into initial state in which each component delegate is separated from each others. Two other states concern two important operations on routing: virtual binding and real binding. After a routing completes the operation of virtual binding, it goes into bound state which means routing is connected and is ready for request. If a request is dispatched to a specified routing, the routing goes immediately into active state in which sequences of real binding and unbinding operations of components are performed. During active state all relevant component instances are loaded into routing, executing its functionality and are unloaded from routing respectively and continuously. Figure 4.18 shows the routing execution in different states. In the following subsections, two important execution operations: virtual binding and real binding, and the procedure of execution are described.

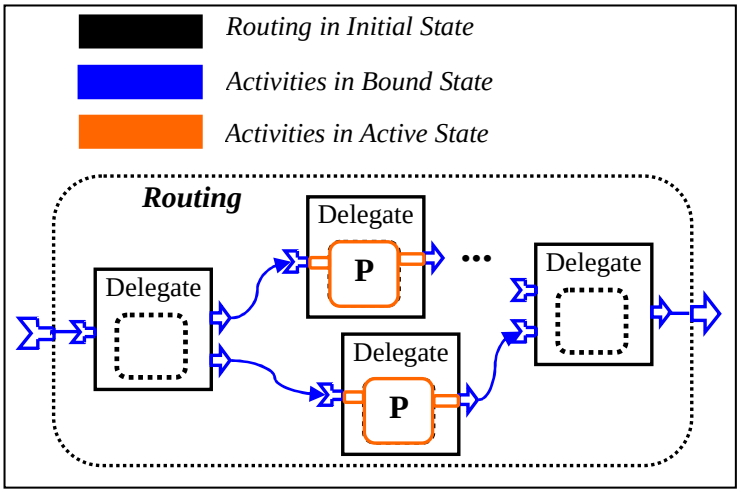

Figure 4.18: Different State of Routing Execution

Virtual Binding

Virtual binding aims to connect a routing, so that the intermediate data can be transferred among component delegates. As shown in Figure 4.19, virtual binding has established the pipeline of dataflow among different component delegates. To establish above dataflow pipeline, virtual binding has to carry out three tasks: IO behavior checking, pre-execution checking, and ports binding.

IO behavior checking is the first task of virtual binding to check whether the IO behaviors, i.e. in/out ports, of component delegate is matched with that of its represented component processor. The checking issues primarily concern whether the port types of two ports are the same and whether the transferred data are for the same purpose. To implement this kind of checking, component delegate temporarily loads a component instance to make a real test for IO behavior checking, and then returns back the component instance after test.

The second task, pre-execution checking, is to check whether the local resource and environment meet the execution requirement of component. The specification of requirements and implementation of pre-execution are implemented in component processor by designer. What the virtual binding need to do is to invoke the common method when component instance is temporarily loaded for IO behavior test.

The third, also the most important task for virtual binding is ports binding which means connecting component delegates together according to the provided or automatically detected binding pairs of coupled ports. From the introduction of communication ports, we get to know, each port has a set of source objects and a set of destination objects. In a pair of coupled ports, there are a supplier and a consumer. Supplier is set as one of sources of the consumer port and consumer is set as one of destinations of the supplier port. That is what need to be done for ports binding. If all ports of all delegates in a routing have been bound, a dataflow pipeline of routing from input set to output set has been created.

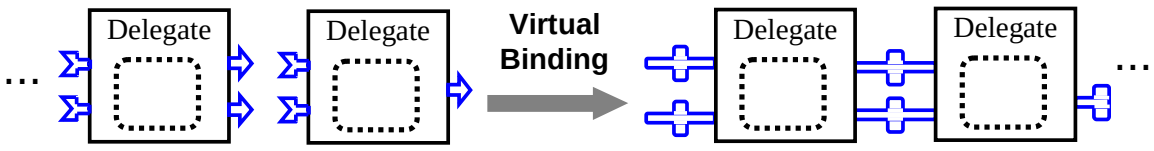

Figure 4.19: Virtual Binding of Routing

Real Binding and Unbinding

Virtual binding has established the dataflow pipeline between delegates. However, after virtual binding the connection between component delegate and its represented processor is still separated which is the responsibility of real binding. For component delegate and component processor, source objects of the in port and destination objects of the out port are involved in process with component delegates, so we call these as outer behaviors of port. Likewise, destination objects of the in port and source objects of the out port are involved in process with inner processor or the inner object of processor, so we call these as inner behaviors of port. During the virtual binding the outer behaviors of delegates have been bound to form dataflow pipeline. But after virtual binding, the inner behaviors of delegates are still empty. The inner behaviors of component processor have already been bound and fixed when it is created and instantiated. One of key tasks of real binding is to bind the inner behaviors of delegates to the outer behaviors of component processors.

During the virtual binding it does not enforce that all in/out ports of delegates have to be bound together. For binding demanded ports, they have to be bound otherwise an exception will be thrown out. For binding optional port, a default value has to be provided for the case that the port is not bound with others after virtual binding. The second task of real binding is to set the unbound port with its provided default value. This step is indispensable. Otherwise routing will be blocked because unbound ports will be waiting for the inputs that never arrive. After the real binding, the routing can be said really ready for request processing. For a component processor, only when it is its turn to execute, it then carries out the real binding to load component instance and executes its functionality. After execution, the processor should also carry out the operation of unbinding from routing.

The real binding of component instance is triggered by data arrival on any in ports of the corresponding component delegate. The unbinding of component is gradually and automatically accomplished by unbinding of each out port that is carried out after out port sends out the result value to next component delegate. Through the temporary binding of component to routing, component is able to serve in several different active routings at the same time. In this case, synchronization is made to ensure different routings can acquire an independent time slice of execution. If necessary, multiple instances of each component can be instantiated to deal with mass requests.

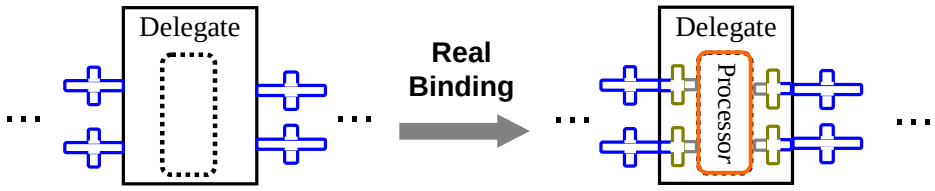

Figure 4.20: Real Binding of Component

The separation of virtual binding and real binding increases the independence of component from others, and also create a complete idle execution environment for all involved components - virtual bound routing which is always in *safe state* on which the dynamic reconfiguration operations can be carried out.

Execution Procedure

The execution procedure of routing can be divided into two stages: routing preparation and request execution. A diagram of execution procedure is illustrated in Figure 4.21. The preparation of routing is rather time-consuming. Fortunately, it only occurs and completes during the initialization of RBW, and the concrete steps are explained as follows:

- *Firstly, parsing routing from XML based Schema to Language based Schema:* XML based schemas are specified and stored in configuration file using a XML based configuration language, refer to section 4.5. The configuration has to be

firstly parsed into language expressed data structure which is easy and quick to instantiate and manage.

- *Secondly, instantiating a routing from schema data structure:* the task of routing instantiation mainly comprises of instantiation of component and creating routing structure for involved components. To be more specific, routing instantiation comprises of instantiating the component processor, creating and initializing the component container, component delegate and communication ports for each component processor, and finally constructing a routing to organize all involved component delegates.

- *Thirdly, making virtual binding on each instantiated routing*, and then creating a routing container for each virtual bound routing, with which Execution Engine can easily get or release a virtual bound routing.

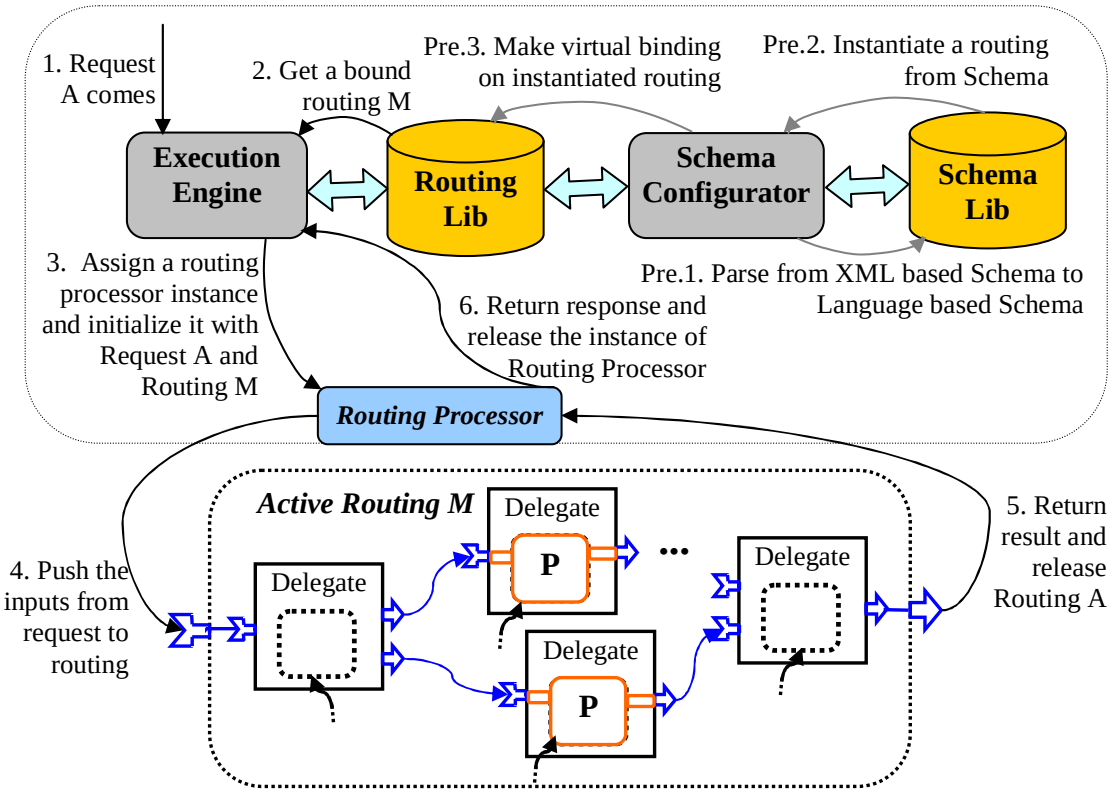

Figure 4.21: Request Execution Steps

The second stage of execution focuses on dealing with the request and is summarized in the following steps:

1. Accepting a request for execution. This request comes from the system in which RBW is integrated, and is reformatted as routing-based request when it is going into the part of RBW.

2. Getting a bound routing. Each request contains a routing ID to indicate which routing is expected to deal with this request. Execution Engine gets the routing according to the specified routing ID from Routings Lib.

3. Assigning a routing processor instance. Routing processor is designed based on thread technology and used to guide and manage the execution of one routing. For each request, Execution Engine assigns a routing processor and initializes it with the accepted request and the fetched routing.

4. Routing processor starts to run and push the input data from request to input set of routing that trigger the execution of routing.

5. After automatic execution of routing, routing processor gets result of the execution and releases this bound routing.

6. Routing processor creates and sends out a response based on the result and the information provided in the request.

7. Execution Engine finishes the execution and releases this routing processor.

4.3.2 Dependency Management

To efficiently manage the execution of components, the dependencies of component have to be recorded and managed. There are two kinds of runtime dependencies of a specific component with its collaborated ones. The most important is dataflow dependencies which specify the data flow relations and are realized by communication ports. The other is control dependencies that specify the control and management relation of one component with its neighbors.

Control Dependency

In RBW, the entity to implement component's business function is component processor whose life cycle management is controlled by component container. Component delegate represents component processor in a routing for collaboration management activities. So the runtime dependencies of component are recorded and controlled by component delegates and control links. The concrete tasks of dependency management in RBW are dependency creating and control events management.

Four methods for dependency creating are listed in Figure 4.22: *register_child()* and *register_parent()* are used to register specified component as the child or parent relation with current one; *unregister_child()* and *unregister_parent()* are used to delete the control dependencies of specified component with the current. Component delegate and control link create their control dependencies according to their data flow dependencies. As introduced before, any component delegate and control link contain in ports and out ports. If the coupled port of an out port P belongs to delegate M, then the owner delegate of port P is a parent delegate of delegate M. Likewise, if the coupled port of in port Q belongs to delegate N, then the owner delegate of port Q is a child delegate of delegate N. According to above definition, the control

```
public class ComponentDelegate implements EventListener{
    ...
    public void register_child(ComponentDelegate delegate);
    public void register_parent(ComponentDelegate delegate);
    public void unregister_child(ComponentDelegate delegate);
    public void unregister_parent(ComponentDelegate delegate);
    public void event_from_child(ControlEvent event);
    public void event_from_parent(ControlEvent event);
    ...
}
```

Figure 4.22: Methods for Control Dependency

dependencies of a component are automatically generated according to the information from its dataflow dependencies. Component may have multiple parent delegates and multiple child delegates because it may have multiple in ports and multiple out ports, as illustrated in Figure 4.23. A component delegate is even possible to become parent delegate or child delegate of itself when it is in a cycled routing and consuming the data produced by itself. After generation of control dependencies for all relevant components and control links, any control event or message can be easily transferred from one component delegate to another in the routing.

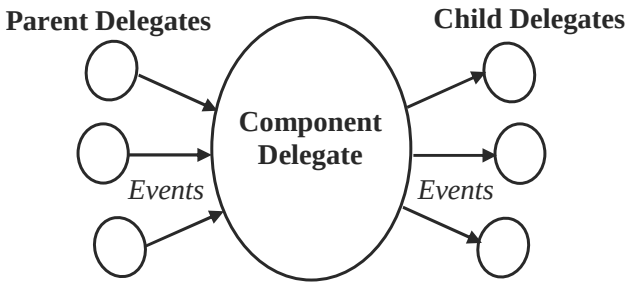

Figure 4.23: Control Dependencies of Component

Another two methods are *event_from_child()* and *event_from_parent()*, that are used to transfer the control events among component delegates and control links according to the created control dependencies. In RBW, six control events are defined. Any event may be triggered automatically in the program or by administrator, and it will be immediately acted on the current component and then broadcasted to all other components in the routing if necessary. Six events are described as follows:

- EVENT_SUSPEND indicates the routing will be suspended. This event can not suspend the started execution of component, but it will suspend the data transferring to next component by suspending the ports of component delegate.
- EVENT_RESUME resumes the suspended routing arose by EVENT_SUSPEND. After receiving this event, component delegate will resume the data transferring among ports and then trigger the execution.

- EVENT_STOP enforce the execution stopped, and throw an exception that will be captured by routing processor and lead to exception handling. This event will not be broadcasted to other components because it is not necessary.

- EVENT_DESTROY results in destroying operation such as stopping the execution and releasing occupied resources etc. on the routing. After destroying, routing will not be returned to the repository.

- EVENT_SUCCESS will be broadcasted to all the component delegates after every successful execution of routing. In some routings, such as flow pick routing, partial component delegates will be blocked even if the execution of routing succeed and is completed. This event is used to unblock such delegates and clear up the routing for next processing.

- EVENT_FAILURE is generated when there is unexpected error. This event will throw out an exception and lead the routing processor to the exception handling procedure.

4.4 Routing Dynamic Change

The most important advantage of RBW is its capability of dynamic change with minimal disruption. During the routing execution, the real binding is separated from virtual binding which generates a complete idle execution environment - bound routing. The bound routing is always kept in safe state in which there is no execution activities of components. So the dynamic change operations can be acted on bound routing without any delay, and results in a new collaboration relation of components. After changing, the routing lib will be updated with the changed routing. The time for updating which is an extremely small value, is the real blackout time to delay subsequent requests. In the following subsection the procedure of dynamic change and the dynamic capabilities are introduced in detail.

4.4.1 Dynamic Change Procedure

In RBW, routing represents the structure and the collaboration relation of software components. So changing a routing means changing the structure of a component oriented software system. Figure 4.24 describes the detailed steps to make changes on a routing in which all requests are assumed to be executed by the same routing:

0. Request A is being executed by Execution Engine with specified routing M. Actually, the concrete task of execution management is assigned to a thread based routing processor.

1. During the execution for request A, a command comes to make change on routing M which is being executed. This command may come from local administrator or from another being executed request.

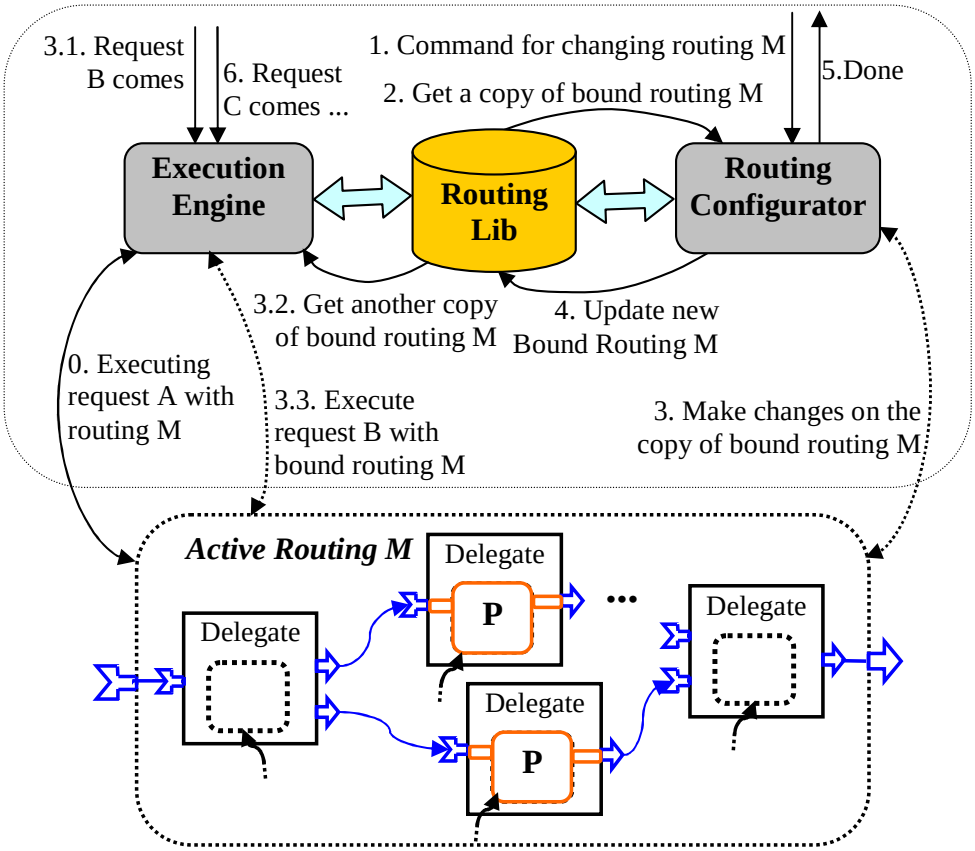

Figure 4.24: Dynamic Change Steps

2. Routing Configurator accepts the command for change, and acquires a copy of bound routing M from repository - the Routing Lib.

3. Routing Configurator makes changes on the copy of routing M. In Figure 4.24 the dotted arrow is used to indicate the being changed routing M and the being executed routing M are different copies. During the change of routing M, the following steps may also happen:

   3.1. A request B comes for execution with routing M.

   3.2. Execution Engine gets another copy of bound routing M from repository.

   3.3. Execution Engine assigns another routing processor instance to execute this request B with new copy of bound routing M.

4. Routing Configurator finishes the change on routing M, and updates the Routing Lib with this changed bound routing M. This updating has to be synchronized with continuous getting operation from Execution Engine.

5. The command for change is done and the changed routing M is available for subsequent requests.

From above introduction, we get to know that, almost at any time, Execution Engine is able to continuously execute requests with a routing or make changes on the same routing. The difference is that the old routing will be used for requests before the routing updating and the new routing will be adopted after the updating. During the procedure of dynamic change, there is no any operation directly acted on components. So the traditional hard issue of consistency preserving is simplified to synchronization of routing updating of routing repository, which can be addressed much more easily and can be always guaranteed to complete in a extremely short time, refer to chapter 7 for details.

4.4.2 Dynamic Capabilities

Operations for dynamic change in RBW can be classified into two categories: routing level changes and component level changes, as listed in Figure 4.25. Routing level changes contain operations for cloning a routing and removing a routing. The operation for modifying a routing is not offered because modification on a routing can be achieved by the combination of component level changes.

Component level changes contain operations, for example, inserting/removing a component, disabling/enabling port, and insert/removing ports binding. When removing a component, the real action is to remove a component delegate from current routing. Only when the component is not used in any routing, the component delegate and its processor will then be deleted from repository. Because ports of component processor are fixed, so the change operation on port is actually to change the ports of component delegate. When disabling a port of a delegate, it means to hide the port of delegate from other delegates and this disabled port will not be bound to any other ports. If an in port is disabled, component processor will adopt its default value as the input when necessary for execution. Only after a port is disabled, the operation of enabling for this port is available. Another important kind of operations are inserting/removing the ports binding pair which are used to adjust the data flow relation. It is especially indispensable when a component is inserted into/removed from the routing.

In addition, there are some operations which change the identifier or property of the routing, component and port etc.

After changing a routing, the new changed routing has to be checked wether it is valid or not. If the changed routing is not valid, the changes on routing are failed. For routing level changes, it is not necessary to make the validation. However, for component level changes, it makes no sense to check the validation only after one single operation. Two transitional operations: *transaction_begin()* and *transaction_end()* are designed to enable a collect of change operations atomic. In programming, invocations of all collected operations between *transaction_begin()* and *transaction_end()* behave as one single operation. The *transaction_begin()* creates an object of *Transactor* to manage the transactions and enables the subsequent operations to act on the same routing. Each change operation belonged to component level has to follow behind the operation of *transaction_begin()*. The collected single operations carries out the change processing without validation checking. The *transaction_end()* is re-

```
public interface RoutingConfigurator{

    public boolean clone_routing(String originalID, String newID);
    public boolean remove_routing(String routingID);
    public boolean transaction_begin(String routingID);
    public boolean transaction_end();
    public boolean insert_component(String componentID);
    public boolean remove_component(String delegateID);
    public boolean enable_port(String portID);
    public boolean disable_port(String portID);
    public boolean insert_ports_binding(String sourceID, String targetID);
    public boolean remove_ports_binding(String sourceID, String targetID);
    ...
}
```

Figure 4.25: Key Operations for Dynamic Change

sponsible to check whether the routing changed by collection of operations is valid or not. If the changed routing is valid, the *transaction_end()* is also responsible to update the *Routing Lib* with the new changed routing.

Additional to the dynamic change, another important advantage of RBW is the multi-solutions supporting. In RBW, the component instances do not fix its interaction and collaboration relations which are modeled by routing. Two different routings represent different interaction and collaboration solutions for components. From previous introduction of routing execution, it is easy to know that different routings stored in the Routing Lib are always available to serve different requests. Component instances are just temporarily loaded to a routing for execution. There is no difference whether a component instance is temporarily to the same routing or another different routing at different time slice. So, the feature of multi-solution supporting is naturally offered to enable developing personalized services or even personalized solutions for system configuration.

4.5 XML based RBW Schema

The offline configuration of RBW is specified by an XML based configuration language whose grammar is defined by XML Schema. XML Schema has powerful express capability, and shares the same expression format and variety of data type with XML [27]. In RBW XML based configuration language contains four parts: port schema, component schema, control link schema and routing schema. The complete meta-definition of the configuration language can be referred in appendix A.

4.5.1 Port Schema

A reduced meta-data definition of port schema is described in Figure 4.26 expressed in XML Schema language. In addition to the optional sub-element of *description*, the meta-data of port contains three exclusive sub-elements: *ioDirection*, *portType* and *usage*, and three attributes elements: *identifier*, *classType* and *defaultValue*.

```
<rbw:complexType name="port">
   <xsd:sequence>
     <xsd:element ref="description" minOccurs="0" maxOccurs="1" />
     <xsd:element ref="ioDirection" />
     <xsd:element ref="portType" />
     <xsd:element ref="usage" />
  </xsd:sequence>
  <xsd:attribute ref="identifier" />
  <xsd:attribute ref="classType" />
  <xsd:attribute name="defaultValue" type="xsd:any" minOccurs="0" maxOccurs="1" />
</ rbw:complexType>
```

Figure 4.26: Port Meta-Data in XML Schema

The sub-element of *ioDirection* indicates the owner port is an in port or out port. So it adopts a data type of enumeration and contains only two possible values: "*in*" or "*out*". The *portType* also adopts the enumeration type with two values: "*operation*" and "*stream*" to distinguish the data passed by this port is an operation or stream data. Similarly, the sub-element of *usage* contains two potential values: "*optional*" and "*required*" that point out whether this port is used in a mandatory manner or not. When the *usage* is set a value of "*required*", it means that this port can not be disabled and has to be bound to another ports. Otherwise, an exception will be thrown out. If a port is set a value of "*optional*" in the *usage*, it means the port is possible to be disabled.

The first attribute element is identifier which is an extension of string with restriction pattern value of ".+#.+" expressed in XML schema regular expression [9]. The part before separator "#" indicates the name of component or link that holds this port. The part after "#" indicates the real meaning for the port. For example, a value of identifier could be "*Authorization#Role*". The second attribute element is *classType*, which is used here to indicate the data type of passed data. The last attribute element is the optional *defaultValue* that is closely related with sub-element of *usage*. When the usage has the value of "*optional*", the attribute element of *defaultValue* has to set a value. Conversely, if the usage has the value of "*required*", the *defaultValue* could be omitted.

4.5.2 Component and Control Link Schema

The definition of component schema concerns three aspects: component delegate and container and processor. As illustrated in Figure 4.27, meta-definition of component schema comprises of three key sub-elements: *parameter*, *property*, and *port*, and two attribute elements: *name* and *classType* that is similar to port schema.

The sub-element of *parameter* is used to instantiate a component processor that contains parameter information in its constructor. Because parameters are not indispensable for every component constructor, it could be omitted. As shown in Figure 4.28, the sub-element of parameter is based on normative type of any, and takes with necessary attributes, such as *name*, *classType* etc. The sub-element of *property* is used both in component container and processor. For example it can be used to

```
< rbw:complexType name="component">
  <xsd:sequence>
    <xsd:element ref="description" minOccurs="0" maxOccurs="1" />
    <xsd:element ref="parameter" minOccurs="0" maxOccurs="unbounded" />
    <xsd:element ref="property" minOccurs="0" maxOccurs="unbounded" />
    <xsd:element ref="port" minOccurs="2" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute ref="classType" />
</ rbw:complexType>
```

Figure 4.27: Component Meta-Data in XML Schema

define the pool size or other application-specific value etc. The last sub-element of *port* is only used for component delegate. At least one in port has to be defined to encapsulate the IO behaviors of component.

```
< rbw:complexType name="parameter">
  <xsd:extension based="xsd:any">
    <xsd:attribute name="name" type="xsd:string" />
    <xsd:attribute ref="classType" />
    <xsd:attribute name="isArray" type="xsd:boolean" />
  </xsd:extension>
</ rbw:complexType>
```

Figure 4.28: Meta-Data of Parameter Element

The schema of *control link* is defined to customize the control link as one of four concrete types: ADDLink, ORLink, XORLink and MapLink. The meta-definition of control link schema, shown in Figure 4.29, comprises of two sub-elements: *linkType*, *port*, and one attribute element: *name*. The sub-element of linkType is extension of string with restriction of four enumeration values of type. The sub-element of *port* is the same to *port* in component schema, but the configuration of ports has to comply with *linkType*, e.g. ports of pick link are fixed with the data type of Boolean.

```
< rbw:complexType name="controlLink">
  <xsd:sequence>
    <xsd:element ref="linkType" minOccurs="1" />
    <xsd:element ref="port" minOccurs="2" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" />
</ rbw:complexType>
```

Figure 4.29: Control Link Meta-Data in XML Schema

4.5.3 Routing Schema

Routing schema is responsible to collect all relevant components and control links for components integration solution or services composition. As shown in the following

Figure 4.30, the meta-definition of routing schema comprises of five key sub-elements: *inputs*, *outputs*, *import*, *controlLink* and *connectors*.

```
< rbw:complexType name="routing">
  <xsd:sequence>
    <xsd:element ref="description" minOccurs="0" maxOccurs="1" />
    <xsd:element ref="inputs" />
    <xsd:element ref="outputs" />
    <xsd:element ref="import" minOccurs="1" maxOccurs="unbounded" />
    <xsd:element ref="controlLink" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="connectors" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" />
</ rbw:complexType>
```

Figure 4.30: Routing Meta-Data in XML Schema

The first two important sub-elements are *inputs* and *outputs*, which create the IO behaviors of the routing by defining two set of NamedObjects. These IO NamedObjects will also be created and bound with ports of components delegate. The meta-definition of NamedObject element has two sub-attribute elements: *identifier* to binding with port, and *classType* to specify the type of stored data. The meta-definition of *import* comprises of only one string type sub-element that is used to specify which component will be imported in routing. If several components need to be imported, the *import* will be used several times with different components. The instance of control link will be shared by several routings. The *controlLink* is used to specify which control links will be used in current routing. The last sub-element is *connectors* that is used to configure how the components, control links, inputs and outputs are connected. The sub-element of *connectors* comprises of multiple *links*, and each *link* specifies one connection between two ports or NamedObjects. The meta-definition of *link*, illustrated in Figure 4.31, comprises of two key sub-elements: *source* indicating the source object of connection, and *target* indicating the target object of connection. Both *source* and *target* use the type of *identifier* to specify a pair of ports.

```
< rbw:complexType name="link">
  <xsd:sequence>
    <xsd:element name="source" type="identifier" />
    <xsd:element name="target" type="identifier" />
  </xsd:sequence>
</ rbw:complexType>
```

Figure 4.31: Meta-Data of Link Element

4.6 Extension for Distributed Components

In previous sections, a complete description of RBW from structure, execution, management to schema configuration has been introduced. However, the introduced RBW

is only applicable for the components deployed in the local computer. In Internet oriented applications the components are also frequently deployed in remote computers or other devices. In order to adapt RBW for such distributed components, an extension of RBW is proposed by extending component delegate.

4.6.1 Extension of Component Delegate

In the extension of RBW, the component delegate is extended into two parts: remote delegate and local delegate, as illustrated in Figure 4.32. The remote delegate represents distributed component in a routing managed by workflow manager. The local delegate represents the component in the computer where the component is deployed. In addition to inherit the responsibility of component delegate, the remote delegate and local delegate have to take charge of the communication of intermediate data over Internet. In the routing of RBW extension, the composed elements are remote delegates, not local delegates. Remote delegate is responsible to take the role of constituted element for routing, manage the dependency among components, and deal with communication for data transferring over Internet etc. The local delegate is responsible to cooperate with remote delegate for data transferring over Internet, and load component instance for execution etc. Since the execution and dependency management of routing have already been introduced in previous sections, here I only introduce the additional functionality concerning communication of intermediate data over Internet.

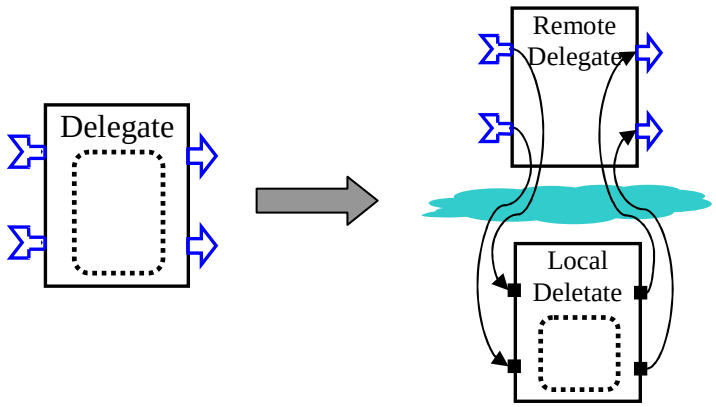

Figure 4.32: Extension of Component Delegate

Named Object Transfer Protocol - NOTP

As introduced in section 4.2, all intermediate data transferred in routing are packed in a uniform wrapper - Named Object. In the extension of RBW, the intermediate data transferred between remote delegate and local delegate are also a series of Named Objects. To efficiently transfer the Named Objects over Internet, a simple Named Object Transfer Protocol is designed. The transfer protocol is designed in application level, and can be implemented on top of other existing communication protocols,

such as HTTP etc. In our prototype, it is directly implemented on top of TCP/IP. The design of NOTP protocol adopts the pattern of request/response. However, the request and response share the same message format because both request and response are used to transfer Named Objects: request is used to transfer Named Objects for inputs of component processor and response is used to send back Named Objects as the result of component execution.

```
<Header>
        <Content-Type> NamedObject </Content-Type>
        <Data-Encoding> http://www.w3.org/TR/xmlschema-2/ </Data-Encoding>
        <Object-Number> ... </Object-Number>
        <Keep-Alive> true/false </Keep-Alive>
        <Version> 1.0 </Version>
</Header>
<Body>
        <Object-Code> ... </Object-Code>
        <Object>
                <Identifier> ... </Identifier>
                <Data-Type> ... </Data-Type>
                <Data-Value> ... </Data-Value>
                <State> ... </State>
        </Object>
        <Object> ... </Object> ...
<Body>
```

Figure 4.33: Request/Response Message Format in NOTP

As illustrated in Figure 4.33, the message of protocol comprises of a header block and a body block. The items of header block are explained as follows:

- *Content-Type*: has to be specified with the text of "Named Object" which is the flag to indicate that subsequent contents will follow the regulation of NOTP.
- *Data-Encoding*: specifies the encoding method used in data expression of NamedObject in the body block. The default value is the namespace of XML schema *http://www.w3.org/TR/xmlschema-2/* to indicate the data type and value are complied with the regulation of XML data expression.
- *Object-Number*: indicates how many objects is scheduled to be transferred in the body block.
- *Keep-Alive*: indicates whether the connection will be kept alive after transferring of one session. Here a session is defined as one request and one response transferring for one execution of component. However, in local delegate and remote delegate the orders of sending request/receiving response are different.
- *Version*: indicates the current version which is designed for future updating.

The body block of protocol comprises of an Object-Code and a series of number-specified Named Objects. The Named Object is transferred by transferring its four constituted elements, which can be used to reconstruct the same Named Object in

another side. Because remote delegate can not catch any control information or even exceptions from local delegate and component processor, an Object-Code is designed to transfer the management information. For all the management information, such as failure, exception etc., can be predicted and is designed in advanced, the Object-Code is simply designed as a four digit number for the sake of efficient transferring. The information that need to be expressed by Object-Code are:

1. The set of Named Objects are from in ports of remote delegate or from out ports of local delegate?

2. This operation is a normal request execution or a test request to know whether the remote delegate is matched to component processor or not.

3. The named objects are the result of successful execution of component or NULL value for failed execution?

4. Which kind of exception is caught during the execution in local delegate?

Interactions between Remote Delegate and Local Delegate

Protocol just specifies the format of transferred message. The interaction model explains how the remote delegate exchanges data with local delegate, as illustrated in Figure 4.34. Each component has one local delegate deployed in the same computer, and one or more remote delegates deployed remotely. The remote delegates represent the same component in different routings that may be deployed in one computer or different computers. Remote delegate inherits from component delegate, and also inherits the management responsibilities in routing. However, compared to component delegate, remote delegate has the following differences:

1. Remote delegate does not hold a component container for loading component instance. As substitution, a remote Connector is embedded in each remote delegate to deal with the task of communication with local delegate.

2. Remote delegate holds customizable inner named objects bound with its in/out ports. The inner named objects are shared with embedded remote connector.

3. Remote delegates rewrite three methods that are concerned with component container: *real_binding()*, *real_unbinding()* and *check_bindable()*. The *real_binding()* is rewritten to enable remote connector creating connection with distributed component instead of loading component instance. The *real_unbinding()* is rewritten to disconnect with distributed component. For the establishment of internet connection is time-consuming, an option can be selected by offline configuration to keep the connection alive during the runtime. This selection can be informed to local delegate by the item of Keep-Alive in protocol header block. Method of *Check_bindable()* is rewritten to send a test request to local delegate to simulate an execution.

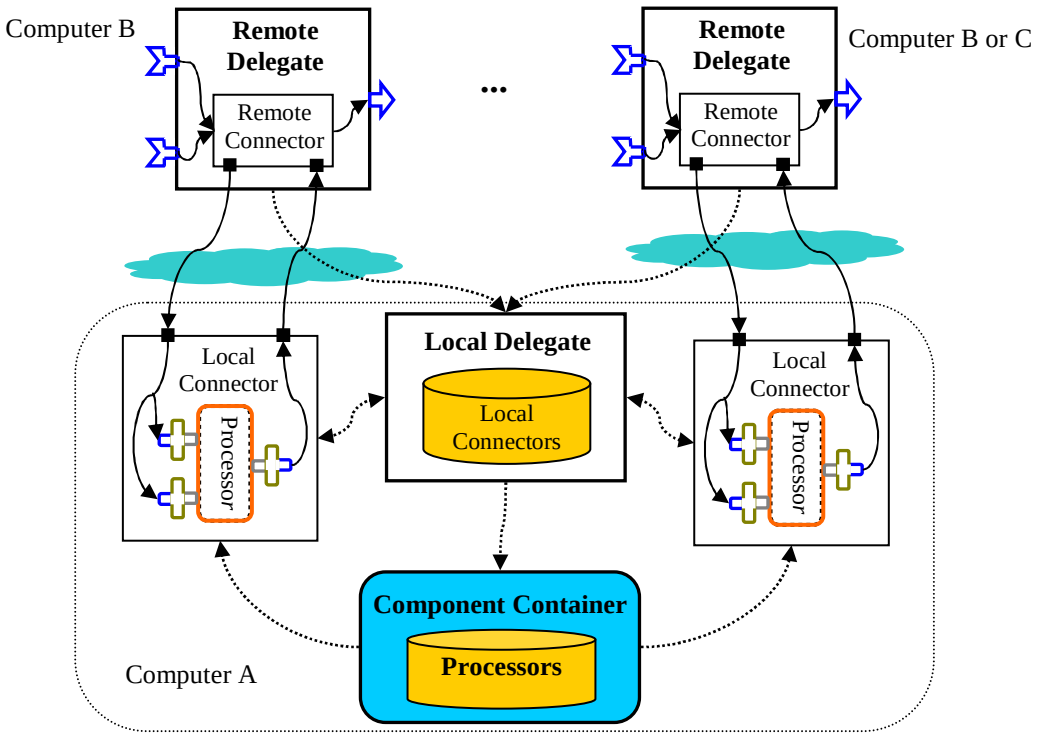

Figure 4.34: Interaction between Remote and Local Delegates

Local delegate is deployed with component together and is responsible to coordinate the communication with remote delegate. Unlike remote delegate, local delegate can not interact with other component processors or other local delegates, and it only interacts with the remote delegate which represents the same component. Local delegate holds customizable named objects as its IO behaviors matched with the represented component processor, and holds the reference of component container. Because the local delegate has to deal with the connection with multiple remote delegates, a repository for instantiated local connectors is designed to be embedded into local delegate. After deploying the component, local delegate starts a listen service for remote delegates. Once a connection is created, local delegate will assign the connection to one idle local connector for concrete connection processing. The tasks of local connector are described as follows:

1. Local connector customizes and initializes itself to be matched with component processor by acquiring the cloned IO behavior of its local delegate and reference of component container.

2. Local connector loads the instance of component processor and binds its IO behaviors with the in/out ports of component processor while receiving a forwarded connection from local delegate.

3. Local connector deals with communications with remote connector and parses the request messages.

4. Local connector pushes the parsed Named Objects to in ports of component processor to trigger the execution of component.

5. Local connector receives the execution results from out ports of component processor, encodes the result into response message, and sends back the response to remote delegate.

6. Local connector also deals with binding test after receiving the test request from remote delegate, and is responsible to translate the test results or execution exceptions into a abnormal response message.

4.6.2 Routing for Distributed Components

In the extension of RBW, the distributed components are virtually organized as a routing by their representatives - remote delegates. As illustrated in Figure 4.35, the components and corresponding local delegates may be deployed in different computers, but all the remote delegates that belong to one routing have to be deployed in the same computer. The interactions of different distributed components are managed in a centralized model. Each component can only interact with other components via the centralized remote delegate, instead of local delegate.

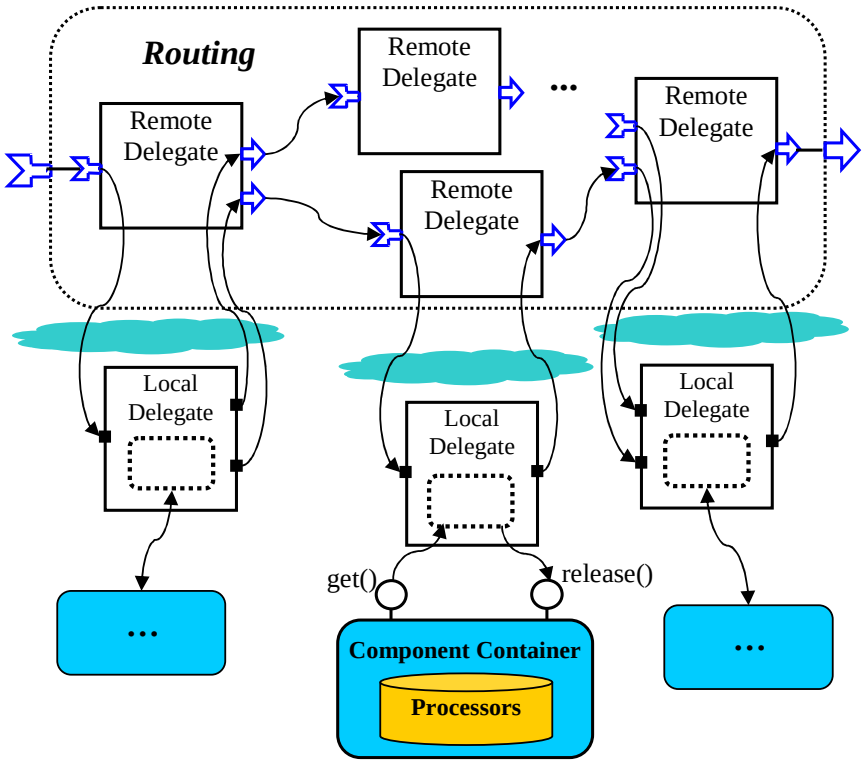

Figure 4.35: Routing for Distributed Components

In the deployment of distributed components, the offline configuration of component schema is the same for remote delegate and local delegate. In fact remote

delegate and local delegate just need to acquire different parts of component schema for initialization. The meta-definition of component schema does not need to change for distributed components. All the additional information can be configured using the sub-element of *property* of component schema. For example, a property item *Keep-Alive* is required to be added in the schemas for distributed components.

Chapter 5

Case Study: Smart Data Server Version 3.0

Middleware is not only demanded to provide powerful functionalities for easily creating distributed applications, but also demanded to have a flexible management for its the functional components to offer adaptive services for varying application environments. As an application case of RBW, a dynamic and secure middleware system - Smart Data Server Version 3.0 (SDS3) is introduced in this chapter. The SDS3 adopts parts of CORBA as the communication infrastructure and creates three secure components to enhance the invocation. Through RBW management on secure components, the SDS3 is able to provide multi-level security control on the deployed applications and the control strategy can be dynamically changed.

5.1 Case Introduction

The project Smart Data Server (SDS) was initiated at Institute of Telematics [81] to create a modular and extensible framework for distributed functionalities that provides a universal and secure accessing to multiple different data sources. Illustrated as in Figure 5.1, the SDS, also called as Smart Data Server Version 1.0 (SDS1), was constructed in a layer-oriented structure: session layer for handling and transferring the request/response, service layer for invoking functionality of application and accessing system service, such as logging, authorizing, database etc., and function layer for deployed applications. To enable communication with other distributed computing technologies, a widely accepted message protocol - Simple Object Access Protocol (SOAP) was integrated into the SDS [44].

To achieve more flexible control and parallel execution of middleware components, we re-encapsulated the IO behaviors of the existing functional components and employed piped workflow for component management to reconstruct the middleware system [45], called Smart Data Server Version 2.0 (SDS2). As shown in Figure 5.2, different modules of SDS2 can be categorized into three layers: the infrastructure layer for core services, the middle layer for piped workflow and managed functional components, and the top layer for applications. In the key part of middle layer, each

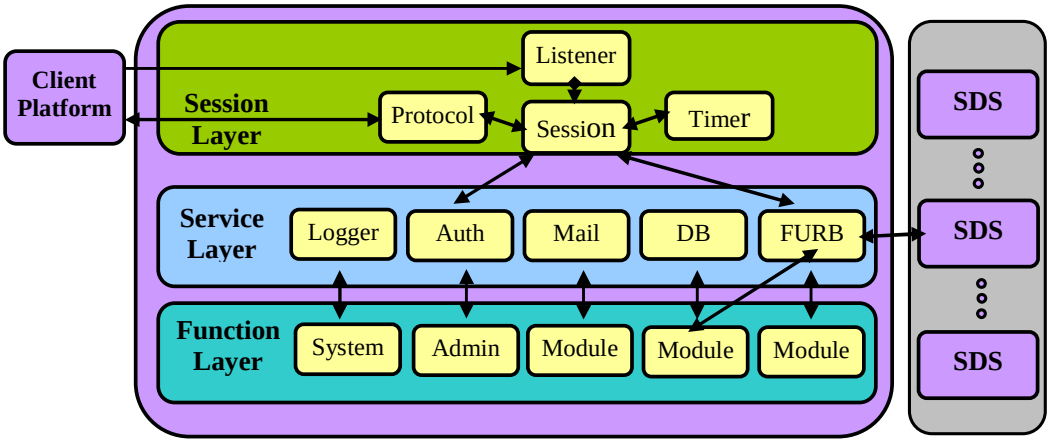

Figure 5.1: Architecture of Smart Data Server Version 1.0

managed component exhibits distinctly its IO behaviors, requirements and properties. To be managed by piped workflow, each component has to implement a specific kind of pipe interface according to the role of this component in the workflow and automatically create a control node as its representative to guide the execution. In SDS2 the execution of components could be processed in a parallel manner, which means part of result could be exported to next component for execution while the current component is still in processing. Another distinguished feature is the flexibility in configuration of components collaboration. Components could be configured and reorganized to offer different middleware solutions for varied applications.

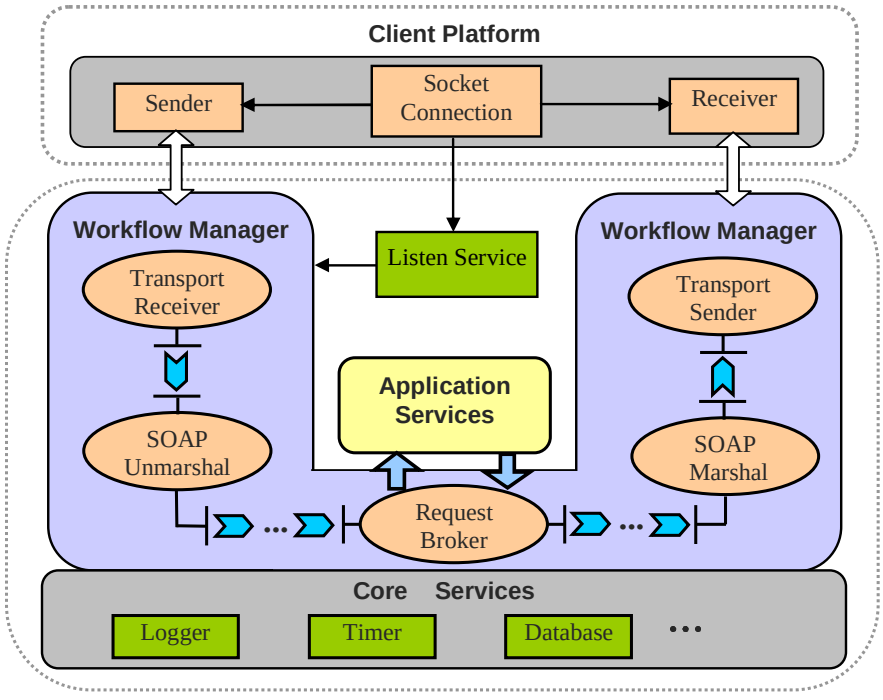

Figure 5.2: Architecture of Smart Data Server Version 2.0

However, the improvements of SDS2 are still not enough to meet requirement of increasing varying application environment. The reorganization of middleware components in SDS2 can only be achieved by making the offline configuration. To meet the demand of dynamic reconfiguration introduced in section 1.1, we create a dynamic secure middleware system - Smart Data Server Version 3.0 (SDS3). In fact the technologies used in SDS3 have no direct connection with former versions of Smart Data Server. The CORBA is adopted and modified as the communication infrastructure of SDS3. The RBW is used to manage the secure components and reach a multi-levels, dynamically changeable security control on the deployed applications. As the infrastructure of SDS3, CORBA also holds the features of dynamics and security which are presented in the following subsections to clarify the differences.

5.1.1 Dynamism in CORBA

The dynamic features of CORBA [40] are involved in the Dynamic Invocation Interface (DII), which defines the client-side interfaces to dynamically create CORBA request, the Dynamic Skeleton Interface (DSI), which is the server-side counterpart to the DII, and the Interface Repository (IFR) etc.

In CORBA the widely accepted means for client to communicate with server is the Static Invocation Interface (SSI) that is provided by static stubs generated by CORBA IDL compiler with the knowledge of target interface at compile time. The DII is a dynamic way to create a CORBA Request to invoke the operations on remote object, however, without relying on static stubs. Three steps are involved to enable invocation with DII:

1. invoking a built-in operation on the object's interface to dynamically construct a Request object;

2. providing details for the request, such as operation name, argument types and values, and the return type;

3. calling an operation on the request object to trigger the invocation, causing the request to be sent to the remote CORBA object.

The DII provides a much more flexible invocation way for the situation that the request information is supplied during runtime. But the performance, e.g. the invocation time, is not so efficient as the way of static stub. This is why most of CORBA applications are written with static stub offered of IDL compiler. However, an important and growing kind of applications, such as interface browsers, network management applications, distributed visualization tools, debugger, and configuration management tools etc., require the dynamism provided by DII.

As the server-side counterpart of the DII, the DSI enables servers to dynamically receive and handle request without compile-time knowledge of operations. Just like in static skeleton each the object implementation inherits from an object-specific skeleton generated by CORBA IDL compiler, in DSI the object implementation has to inherit from a common abstract object of *DynamicImplementation* in which abstract

methods, such as *invoke()* etc., are expected to be overridden. The key class to realize dynamic skeleton in DSI is the *ServerRequest* that contains methods for concrete servant to de-marshal request for execution, as well as to marshal reply to the client. In addition to the flexibility in handling request, there are also two primary tradeoffs compared to static skeletons in a CORBA server:

1. low performance for the type information is discovered during runtime;
2. complexity increasing in programming of DSI object implementation because the programmer has to do extra work to enable exactly invocation, such as de-marshalling distinctly the arguments of request, setting the return value and raising an exception.

The IFR allows application to store and retrieve the type information dynamically, which works like a dynamic IDL compiler. The IFR normally is offered in the way of CORBA services that can be accessed by passing the string "*InterfaceRepository*" to the *ORB::resolve_initial_references* method, or more directly by using the *CORBA::Object::get_interface* method. The usage of IFR is often combined with DII for creating a dynamic client Request and with DSI for parsing type information. The IFR separates the meta-information from CORBA objects and offers a series of methods to access the meta-information. However, the IFR is actually seldom used in practice for complexity of usage and consistency management for meta-information with corresponding objects.

Above dynamic features are only concerned how to dynamically access the CORBA application without the knowledge of type information at compiling time. In our SDS3 the dynamics focus on changing the collaboration relation of middleware components to adapt the variety of application environments and requirements.

5.1.2 Security in CORBA

The CORBA security consists of a set of security models [40], [39] and series of security interfaces [41] for application developer, administrator and implementer etc. The security features contained in CORBA security models focus on the following aspects:

- *Identification and Authentication*: Any active entity, a human user or software system, has to be registered as an identification to establish its right to access objects in the system. The identification may be authenticated in a number of ways while it is involved in the system activities.
- *Authorization and Access Control*: User acquire the permission to do what he/she wants with authorization operations involving the granting, denying, and revocation of access right. The access control is the mean to realize the privilege definition process.
- *Auditing and Non-Repudiation*: Security auditing assists in the detection of actual or attempted security violations, and is achieved by recording details of

security relevant events in the system. Non-repudiation services enable to make users accountable for their actions.

- *Secure Communication*: The secure interoperability can be achieved by three ways: i) the Object Request Brokers (ORBs) share a common interoperability protocol; ii) consistent security policies are in force both at the client and target objects; iii) the same security mechanism is used.

- *Security Administration*: the administration of security activities for an application includes security policy, users, roles and permissions etc.

The CORBA security model is specified in details by offering series of security interfaces from the view of different types of users as follows:

- *The End User View* focuses on the actual, individual principal, the privileges that are authorized, and the authentication that must take place to confirm the identification.

- *The Application Developer View* focuses on the degree that stakeholder (e.g., software engineers, programmer, developers etc.) must be aware of the security capabilities of the enterprise. In some situations the enterprise may wish to defines security, but make those definitions transparent to the majority of stakeholders. In this case, the ORB security services could be automatically called to enforce security of the principals against target objects. In other situations, the security may be the strict responsibility of all stakeholders, who would interact explicitly and programmatically with the security services.

- *The Administrator View* is the security management perspective, providing all of the capabilities to administer and control security policy definition and enforcement, including: creating and maintaining the domains, assigning the privilege attributes to end users, administrating the security policies, monitoring the control attributes of target objects, etc.

- *The Object Implementer View* differs from the application developer's view, since these stakeholders are actually responsible for prototyping and implementing the ORB. The interfaces for this view are intended for the case that a given enterprise decides to implement its own unique security services.

The CORBA security specifications really offer comprehensive security capabilities at the model level. However, it is only a specification without implementation. Further, there is no any product or open source implementation satisfying all the capabilities and features of the CORBA security. In our SDS3 the multi-level security controls are different from CORBA security in the following two aspects:

1. Security capabilities of SDS3 are provided by middleware secure components, not by security services. The high level security control strategy can be changed during runtime for the secure components are managed by our RBW.

2. The Attribute Certificate (AC) are employed in SDS3 to create a security model of Role Based Access Control (RBAC) to enhance the flexibility of roles management used for access control.

5.1.3 Overview of SDS3

In SDS3 the RBW is integrated into a modified CORBA infrastructure to construct a dynamic secure middleware system. In the implementation, an open source Java implementation of CORBA, named openORB [75], has been adopted and modified as the communication infrastructure. Three secure components are created as the middleware core component and work in a model of Role Based Access Control (RBAC) [82]. Instead of as middleware services, here the secure components directly cooperate with Request Broker component to enable secure invocation. For the sake of management by RBW the collaboration relation of secure components can be changed during runtime to adapt the varied application environment.

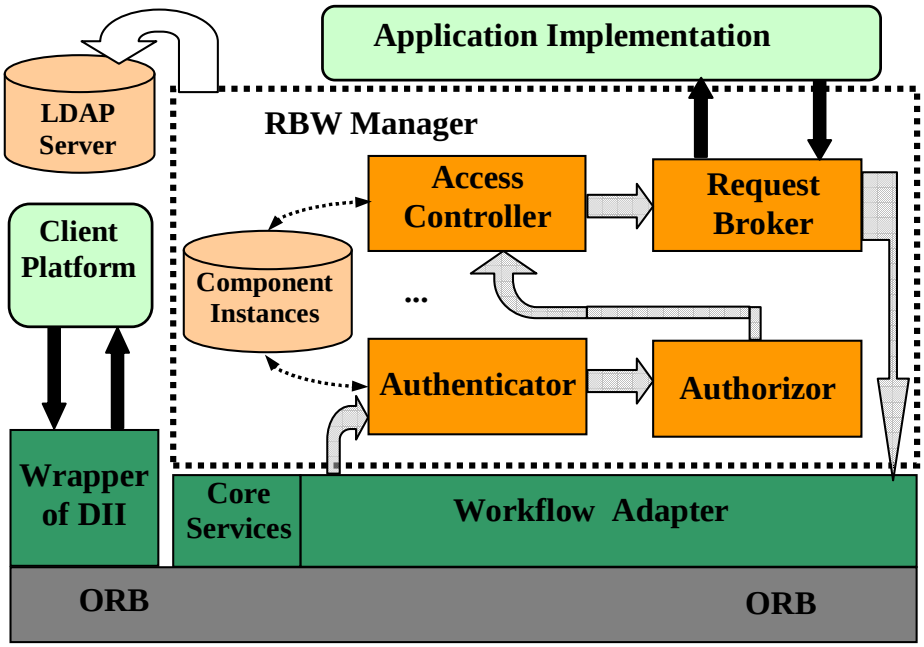

Figure 5.3: Architecture of Secure Middleware - SDS3

The architecture of SDS3 is depicted in Figure 5.3, where not the whole CORBA, but only the Object Request Broker (ORB) is adopted to transfer the request/response messages over Internet. To integrate with RBW, a Workflow Adapter is designed to intercept the request from ORB to RBW or return the response back to ORB. The Workflow Adapter works in a role as Basic Object Adapter (BOA) or Portable Object Adapter (POA) in CORBA. Likewise, the RBW bears the responsibility, similar as Dynamic Skeleton Interface (DSI) in CORBA, to enable the request to invoke the application implementation and to create response based on invocation result. In the client side, the Dynamic Invocation Interface (DII) is wrapped to create routing based request. All the rest parts of CORBA, e.g. DSI, POA, and all kind of

CORBA services etc., are stripped for non-necessity and simplicity. The ORB used in SDS3 is only taken as the communication infrastructure and can also be replaced by other transport protocol, such as Simple Object Access Protocol (SOAP) etc. In SDS3 three security capabilities are offered: Public Key Infrastructure (PKI) based authentication and non-repudiation, Attribute Certificate (AC) based privilege management, and Role Based Access Control (RBAC). Here the security for transport has not been provided because it is a partial task of communication protocol which is not the emphasized point of SDS3, and can be realized by replacing an ORB with the capability of secure interoperability.

5.2 Communication Infrastructure

The SDS3 adopts the Internet-Inter ORB Protocol (IIOP) to define the data format of request/response. Because in CORBA the ORB is designed on top of IIOP and responsible for delivering the request/response which includes message encoding, transferring and parsing, the ORB is also taken as the communication infrastructure of SDS3, presented in the following subsection. Two other functional parts closely related with communication infrastructure are also introduced: a wrapper of DII designed to create routing based request in the client side, and the Workflow Adapter, an implementation of abstract Object Adapter, to take the role similar as BOA.

5.2.1 Original Communication Infrastructure

In CORBA the ORB request/response is transferred by a General Inter-ORB Protocol (GIOP), which can be mapped onto any connection-oriented transport protocol. The specific mapping of GIOP on TCP/IP connection is called Internet Inter-ORB Protocol (IIOP). There are also other mappings on proprietary networks. For example, an Environment Specific Inter-ORB Protocol for the OSF DCE environment is called DCE Common Inter-ORB Protocol (DCE-CIOP). Currently most application of GIOP is dominated by IIOP, also used in SDS3. IIOP is a high-level protocol that takes care of many of the services associated with the levels above the transport layer, including data translation, memory buffer management, dead-locks and communication management. IIOP assumes the client-server computing model in which a client program always makes request and a server program waits to receive request from clients. For a client to make a request on an object distributed in Internet, it must have an object reference to identify the distributed object. Once an application has an object reference, it has all the information it need to connect to the object and make remote invocation on the object's methods. As part of protocol, CORBA specifies a universal format for the object references known as Interoperable Object Reference (IOR). This enables the information about an object reference to be either stored or transferred directly to clients in a form which is universally understood. The information encoded in an IOR consists of the following pieces of information: type of the object, host address, port number and object key. The structure of IIOP can be simply described in the following three aspects:

- *Definition of Common Data Representation (CDR)* specifies a coding syntax for all data types, including basic types, structured types, and object references etc. The CDR coding translates data types into a series of bytes to make up an octet stream.
- *IIOP Message Formats*: IIOP defines seven types of messages to allow client to pass invocations to servers and receive replies which can either be normal or indicate some error status. Typically, each IIOP message contains three elements: IIOP header, message header and message body. The two most important message formats are the Request message used to invoke distributed object by client, and Reply message used to return the response to client from the server. Other message formats are CancelRequest, LocateRequest, LocateReply, CloseConnection and MessageError.
- *Transport Management* give a high level view of the semantics of setting up and ending connections, where the roles of client and server are respectively outlined.

In CORBA IIOP is built in ORB level, which marshals and un-marshals the IIOP requests/responses. The SDS3 is built above the level of ORB and create relevant modules to integrate with RBW managed components.

5.2.2 Wrapper for Dynamic Invocation Interface

In CORBA there are two ways to invoke operations on target object: Static Invocation Interface (SII) and Dynamic Invocation Interface (DII), which are introduced last section. To achieve higher flexibility, the DII is adopted and wrapped in the SDS3 to dynamically construct request for object invocation. For clearly explaining the wrapper of DII, a example for the usage of DII is firstly given in Figure 5.4

```
…
CORBA.ORB orb = ORB.init();
Object object = orb.string_to_object(ior)//obtain object reference.
CORBA.Request request = object._request("get_quote");
request.add_in_arg().insert_string("SAP");
request.set_return_type (CORBA._tc_long);
request.invoke ();
Long retval = request.return_value (); …
```

Figure 5.4: Example of a CORBA Request with DII

In original DII client program the object reference, uniquely identifying the target object and implementing the client stub interface, has to be obtained before request creation. In CORBA there are two ways to obtain the object reference: i) using ORB operation *string_to_object()* in which the argument, string of IOR, has to be created using the counterpart ORB operation *object_to_string()* and be transferred by other means: such as physical duplication; ii) using the CORBA Naming service

which needs the whole CORBA platform. To adapt the DII for SDS3 where only ORB is used for request/response transferring, a wrapper of DII is created. The wrapper of DII enables to directly create request for remote object in client program. But the paid price is that the information about target object, such as host name, port and object name etc, has to be specified distinctly when creating request. In addition, the wrapper of DII provide functionality to conveniently transfer the routing specific information with request together, such as routing ID, signature etc. The implementation of DII wrapper is based on the original DII package, including class of *Request*, *NVList*, *Context* and *Environment* etc. One additional function of the wrapper package is class of *IOREncoder* which provides the functionality to create Interoperable Object Reference (IOR) instance that is necessary to directly create request. In creating IOR instance by IOREncoder, all relevant information, such as type, host name, port number and protocol etc., are distinctly specified as arguments. Based on IOR encoder and original DII package, a routing based request could be created directly in client side. The routing information, such as routing id, signature etc., are encoded into the context of request, which will be parsed distinctly in the server side counterpart - Workflow Adapter. An example of SDS3 request created with DII wrapper is illustrated in Figure 5.5

```
…
IIOPRoutingRequest request =
              new IIOPRoutingRequest(host,port,target,operation);
request.add_parameter(“Hello,“);
request.add_parameter(“you are welcome“);
request.set_routing(“SecureRouting“);
request.set_return_type(String.class.getName());
request.invoke();
String result =(String)request.get_result();
```

Figure 5.5: Example of a SDS3 Request with DII Wrapper

5.2.3 Wrapper for Object Adapter

Object adapter is a mechanism in CORBA to associate a servant with object implementation, activate or inactivate a registered object, and de-multiplexe incoming requests to relevant object implementation. Object adapter normally collaborates with IDL skeleton or Dynamic Skeleton Interface (DSI) to dispatch the appropriate operation up-calling on the implementation of specified servant in request. The Basic Object Adapter (BOA) is the first concrete implementation to realize the mechanism of object adapter. But the BOA was widely recognized to be incomplete and unspecified. For instance, the API for registering servant with the BOA was unspecified. Therefore, different vendors provided their implementation of ORB by making individual interpretations and extensions, which are incompatible with each other. The Portable Object Adapter (POA) was adopted by OMG as the solution to substitute BOA. The POA provides a series of advanced capabilities for object management:

1. the POA allows programmer to construct servants that are portable between different ORB implementation which is the major shortcoming of BOA;

2. the POA enables to support servants that provide consistent service for object whose lifetimes span multiple server process lifetimes;

3. the POA supports transparent activation of objects and implicit activation of servants, which makes the POA easier and simpler to use;

4. the POA allows a single servant to support multiple objects simultaneously, thereby conserving memory resource on the server;

5. policies are associated with servants in POA to enable rich and flexible management on objects etc.

In SDS3 the Routing Based Workflow (RBW) takes the role similar as IDL skeleton or DSI to manage the secure components enabling secure invocation on target objects. To integrate RBW with ORB, a workflow oriented object adapter - Workflow Adapter is designed to take charge of the tasks similar to BOA or POA. In CORBA each object has to be registered to an Object Adapter and activated to be ready for execution. When an object is invoked from client, the hosted Object Adapter acquires request from ORB and helps to access the object implementation. In SDS3 Workflow Adapter is registered as the default Object Adapter for all deployed application objects. Workflow Adapter intercepts request from ORB and forwards it to workflow manager of RBW. The Workflow manager is responsible to assign a routing to process the request and retrieve the response back to workflow adapter through which response is sent back to remote client via ORB. The implementation of workflow adapter is realized by implementing the common interface of object adapter plus additional functionality. The common functionalities offered by workflow adapter are the same to BOA, such as object reference interpreting, servants activating and deactivating, request de-multiplexing, and collaborating with RBW etc. The additional functionality of workflow adapter is to create RBW-specific request from ORB request. In the DII wrapper the routing-specific information, such as routing id, signature etc., are encoded into the context object of original request. Workflow adapter is the server-side parser to decode such information to restore a RBW-specific request used in RBW for secure invocation, as illustrated in Figure 5.6.

5.3 Middleware Components

Five components are provided in SDS3 to realize the secure invocation for request, namely Authenticator, Authorizer, Access Controller, Request Broker and Abnormal Handler. In addition, a control link, AND Link, is used to automatically and dynamically select the flow direction to different components. The indispensable component is Request Broker that directly collaborates with Workflow Adapter to make invocation on application objects. The component of Abnormal Handler is designed to deal with the error, exception and unauthorized accessing etc. Three components of

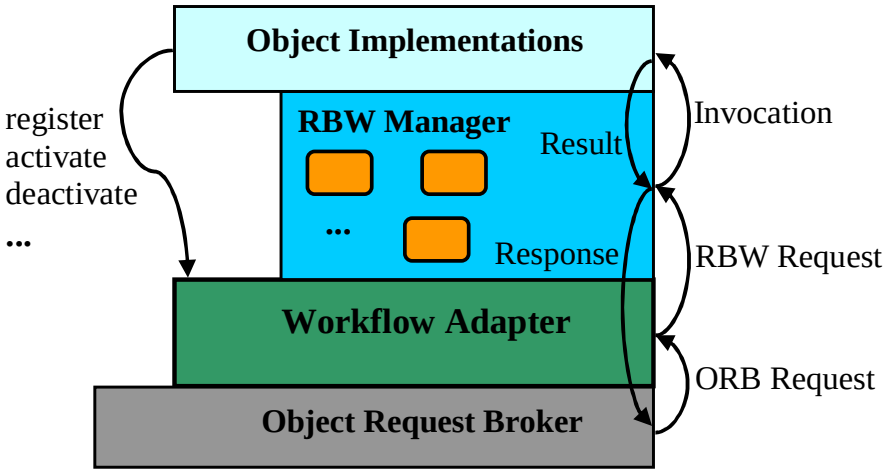

Figure 5.6: Working Mechanism of Workflow Adapter

Authenticator, Authorizer and Access Controller work together in a model of Role Based Access Control (RBAC) [82] to enhance the security of invocation. In our implementation, Public Key Certificate (PKC) based Authenticator is used to identify a user, and Attribute Certificate (AC) based Authorizer is used to express the roles associated with user. A policy is employed in Access Controller component to specify the detailed rules for accessing. As the aided functions, a Security Center is designed and implemented to issue the PKCs and ACs and an open LDAP server is provided to store and retrieve PKCs and ACs issued by our local Security Center. In the following subsections the components management by RBW is first described, then the functionality and design of each secure component are introduced respectively.

5.3.1 Management by RBW

In SDS3 the RBW takes charge of the organization of components and makes secure invocation on application objects. The executions of collaborative components are managed by RBW, whose workflow diagram is illustrated in Figure 5.7. As introduced in chapter 4, each component used in RBW has to inherit from generic processor and create its IO behaviors using communication ports. For example, the component of Authenticator accepts byte stream of signature and outputs identity information, the component of Authorizer accepts the identity and outputs roles associated with accepted identity, and component Access Controller accepts the roles and output a permission to indicate whether the current user is allowed to invoke the specified application. The AND control link accepts the permission result and outputs one Boolean value to pick the next component: TRUE to pick component Request Broker and FALSE to pick component Abnormal Handler. The RBW is initialized when the server of SDS3 starts. During the initialization each concrete component will be instantiated, and for each component one customized component delegate and one exclusive component container will be created automatically to enable the management by RBW. All the information used for initialization, such as

component properties, communication ports and structure of available routings etc., are provided by offline configuration, referring Appendix B.

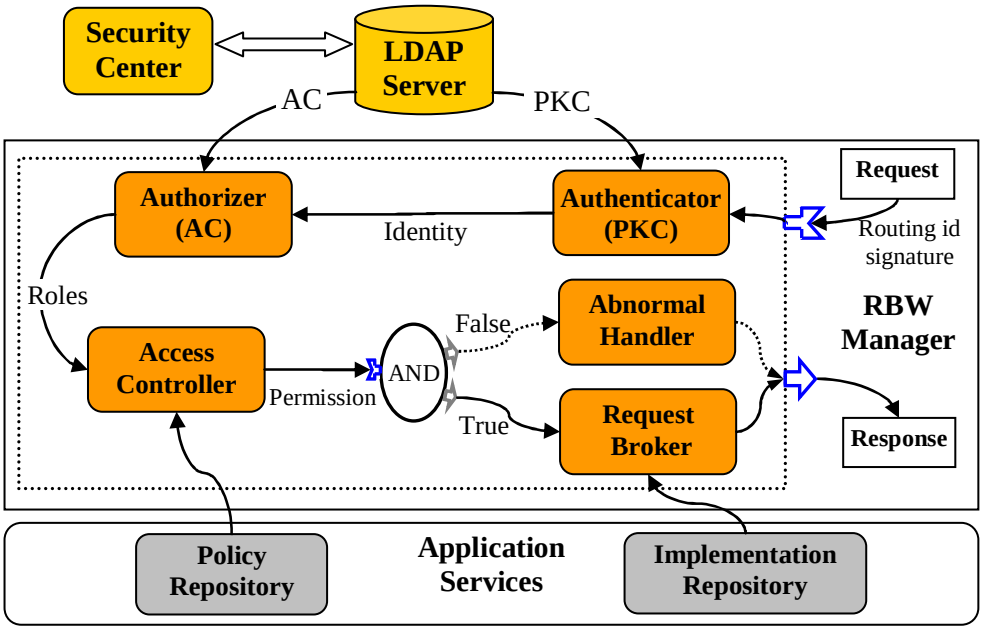

Figure 5.7: Secure Components in RBW

The execution and management inside of routing are already introduced in section 4.3, here we explain the integration with SDS3. To integrate with rest part of SDS3, RBW-specific request and response are designed. The RBW request is parsed and encoded from ORB request by workflow adapter. Addition to regular invocation information, such as target object, operation, parameters etc., there are routing-specific information, such as routing id and signature etc. All these information are extracted and stored in format of NamedObject. For example, the information of signature and target object respectively have the identifiers: "*Request#Signaure*" and "*Response#Target*". During the execution preparation, the RBW request and a specified bound routing will be used to initialize a routing processor. all relevant NamedObjects from RBW request will be set as inputs of routing and trigger the execution of routing. After execution, the response is directly created by methods of RBW request: *write_reply()* or *write_reply_exception()*, and then be sent to client.

5.3.2 Component of Authenticator

The security technology of Public Key Infrastructure (PKI) is employed in the component of Authenticator to realize functions: authentication and non-repudiation.

Public Key Infrastructure

The concept of public key cryptography, also called asymmetric cryptography, was firstly proposed to allow users communicating securely without having a shared key,

by using a pair of mathematically related cryptographic keys, designated as public key and private key [61]. Each public/private key can only decrypt the information encrypted by its corresponding private/public key. The public key is published widely, but still associated with its owner, and the private key is only known to its owner. In a general process of data encryption/decryption with asymmetric cryptography, illustrated as in Figure 5.8, the sender acquires the recipient's public key from a public repository to encrypt the clear data and sends out the result. The recipient decrypts the received scrambled data with his private key, and gets back the clear text.

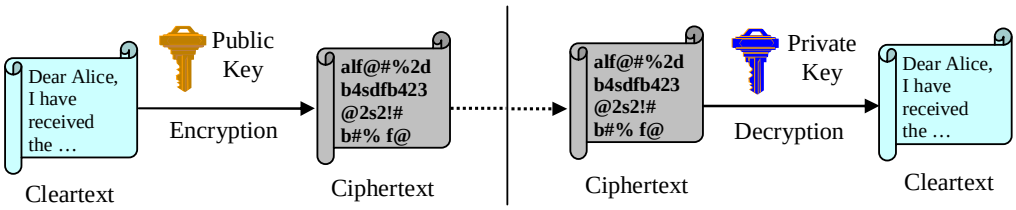

Figure 5.8: Data Encryption/Decryption using Public/Private Key

To make the public key technology applicable in practical application environment, Public Key Infrastructure (PKI) is proposed to bind public keys to entities, enable other entities to verify public key bindings, and provide the services needed for ongoing management of keys in a distributed system. The basic data structure used in PKI is a digital certificate, called Public Key Certificate (PKC), which identifies an individual, a company, or other entity with an associated public key, and is protected by a digital signature of the issuer. As widely used industry standard, X.509 specifies a data format for public key certificate [43] that contains six mandatory and four optional common fields. The mandatory fields are: the series number, the certificate signature algorithm identifier, the certificate issuer name, the certificate validity period, the public key and the subject name that controls the corresponding private key. Four optional fields are the version number, two unique identifiers, and the extensions. Two other important functional component of PKI are Certificate Authority (CA) that acts as a trusted and independent provider of PKC, and Repository that is a database of active digital certificates and is mainly used to confirm the status of digital certificates. The CA has its own published root public key certificate and securely maintained private key. Clients use the root certificate to verify the signatures issued by CA. The basic PKI functions performed by CA are: issuing certificates (i.e., creating and signing them), maintaining certificate status information, and issuing Certificate Revocation Lists (CRLs) etc.

The widely used application of public key cryptography is the digital signature which makes an unalterable mark on the electronic document to ensure the document is originated from the person who signed it. The signature is made in two steps: i) a Hash algorithm is operated on the original document to extract a message digest with fixed length; ii) private key of the owner is used to encrypt the message digest to create signature. Any entity holding the public key of signer is capable to verify the validation of the signature. PKI technology is also used in data encryption/decryption to ensure the data confidentiality. For the reason of time-consuming of the algorithm

execution, PKI is often used to encrypt/decrypt the short message, such as symmetric share key etc.

Design of Authenticator

In the component of Authenticator two functions are actualized: authentication and non-repudiation. Concretely speaking, Authenticator has to recognize the identity of current user and make sure that the identity is not impersonated by other people or entity. The implementation has employed Java cryptography package and IAIK security package [46] which provide all kind of security algorithm implementation, including public key cryptography. To enable authenticate the request from remote client, a Signature Certificate (SC) is designed to carry all related information, such as signature owner and signed signature data etc. The data structure of Signature Certificate contains elements listed as follows:

- *Subject Distinguished Name*, indicates the entity to make the signature within the request.
- *Algorithm Name of Digital Signature*, indicates which algorithm is used for signature.
- *Provider Name of Digital Signature*, indicates provider of implementation of signature algorithm.
- *Signed Data*, is the signature entity which is signed by subject with its private key.
- *Clear Data*, is the original plain text used for signature algorithm.

```
public class SignatureCertificate{
    ...
    public void update_clear_data(byte[] clearData);
    public boolean sign(PrivateKey privateKey);
    public boolean verify(PublicKey publicKey);
    public byte[] to_bytes();
    public boolean from_bytes(byte[] bytes);
    ...
}
```

Figure 5.9: Primary Methods of Class of Signature Certificate

The primary methods realized in Signature Certificate are illustrated in Figure 5.9. In the client side, the method of *sign()* will be used to create a signature on updated clear data. The signature certificate will then be serialized to bytes stream by *to_bytes()* method before transferring with request to distributed components of SDS3. After received the bytes stream of signature, the component of Authenticator will restore the bytes stream to a signature certificate object using method of *from_bytes()*,

and then make verification and extract relevant information. For any secure request before sending in client end, the sender has to apply for a PKC in advance from the security center of the SDS3. In server side the Signature Certificate will be firstly parsed by Authenticator to get user DN with which corresponding Public Key Certificate (PKC) could be retrieved from open LDAP server to make the verification. To be serialized to byte stream in a unique format, each element of SC has to comply with the format as the following Figure 5.10.

Length for Name	Element of Name	Length for Value	Element of Value

Figure 5.10: Serialization Format of SC Elements

As one component managed by RBW, Authenticator exhibits its IO behaviors by unique format - creating fixed communication ports. Authenticator accepts bytes stream of Signature Certificate as its necessary input, and outputs a Distinguished Name (DN) to identify the current user. A Boolean value is also outputted to indicate whether the digital signature is successful to be verified or not.

5.3.3 Component of Authorizer

Different from component of Authenticator to identify who is the entity, the component of Authorizer adopts the technology of Attribute Certificate to make clear what rights and obligations are authorized to the entity.

Attribute Certificate

In the fourth edition of X.509, an Attribute Certificate (AC) [28] is proposed to bind the subject with one or more attributes information, such as authorization etc. AC is taken as the complementation of Public Key Certificate (PKC) to enhance the public key based security control and management. Instead as the extension field of PKC, AC has the similar data structure with PKC, and is also digitally signed by its issuer. There are two reasons to propose AC to make distinctly separation from PKC. Firstly the authorization information stored in AC does not often have the same lifetime as the bound subject identity and public key. Secondly, the PKC issuer is usually not authoritative for the authorization information. A subject may have multiple AC with each of its PKC. Just as the issuer of PKC is called a Certificate Authority (CA), the entity to sign an AC is called an Attribute Authority (AA), and the root of trust is called the Source of Authority (SOA). Similar as X.509 PKC, X.509 AC also has the field to describe the signature algorithm and the signature value. The primary information contained in AC are as follows:

- *Holder* describes the holder of the attribute certificate. The most common way for field is a reference to the holder's PKC via the unique serial number which binds the CA of the PKC and the AC issuer together.

- *Issuer* links the AC to the Attribute Authority (AA) by a single general name which may contain e.g. a Distinguished Name or an IP address of the AA.
- *Serial Number* identifies the AC uniquely from the other ACs issued the AA.
- *Validity Period* states the period for which the AA certifies that the binding between the holder and the attributes will be valid.
- *Attributes* can contain any data to represent the attribute value. The typical types of attribute are role, group, access identity and service authentication information etc.

The widely used application of AC is Privilege Management Infrastructure (PMI) which is also the original purpose of AC. The PMI provide a series of services to enable privileges to be allocated, delegated, revoked and withdrawn in an electronic way. In most applications PMI works closely with PKI to provide privilege management. A PMI is to authorize the entity on which a PKI is to authenticate.

Design of Authorizer

The component of Authorizer is responsible to give the authorization for specified entity. More concretely, Based on inputted identity of the entity, component of Authorizer fetches its Attribute Certificate and extracts the verified roles of the entity. During the implementation an open source project - IAIK security package [46] is used for it provides API for X.509 public key Certificate and Attribute Certificate. Because the AC used in Authorizer component is employed to store the role information, two classes of Role and RoleType are designed to express the role and fulfill relevant functions. The RoleType is rather simple, constructed from a general name and a unique object id. For example, two default role types are set as: "*AuthorRole*", and "*UserRole*". Each role contains the following information: role type, role value, beginning date for the valid period, end date of the valid period, and the holder of role extracted from the Holder field of AC.

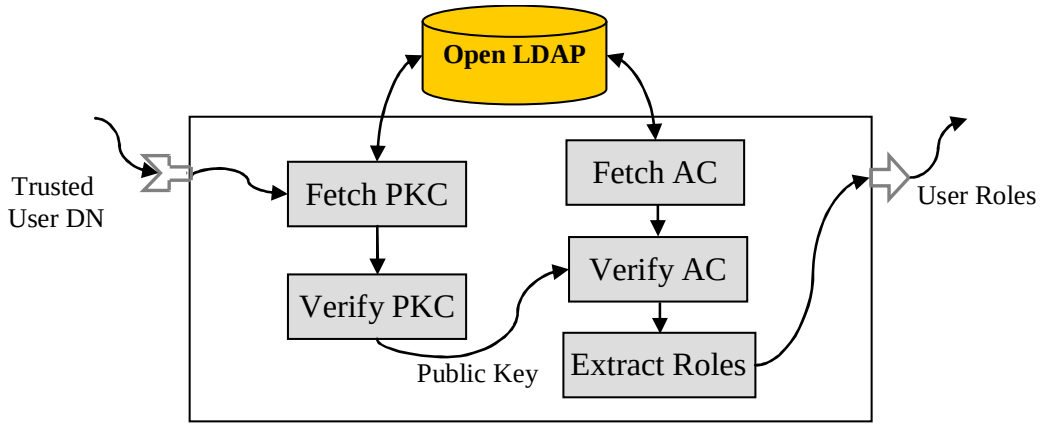

Figure 5.11: Work Diagram of Authorizer Component

Before the accessing every user or entity has to be assigned an AC associated with its PKC via security center, referring section 5.3.5. The AC can be revoked or reassigned at any time according to new requirements. Each AC may contain one or more different type of role, and each type of role may also have multiple values. The working diagram of Authorizer component is illustrated as in Figure 5.11. For component of Authorizer, the input is a trusted Distinguish Name (DN) of current user with which the PKC and AC can be retrieved from open LDAP server. Firstly Authorizer fetches the PKC and then verifies whether the PKC is valid or not. At the same time the AC is also fetched from open LDAP server and verified using the public key extracted from PKC. Finally the Authorizer extracts the roles from AC, and checks whether the date of roles is beyond the valid period or not before outputting to next component.

5.3.4 Component of Access Controller

The component of Access Controller realizes functions for role based access enforcement. It often works with other components, which provide services for authentication and authorization etc., to construct a security enhanced Role Based Access Control (RBAC) for target application invocation.

Access Control Model

Access control refers to a security protection mechanism that controls who is allowed to access the protected specific resource. The access is the operation to do with the computer resource, e.g., execution, view and modification. Ideally, the access control model should be flexible enough to configure all resources with different access constrains, and protect against the unauthorized viewing, modification or execution. There are plenty of accepted access control models, such as Discretionary Access Control, Mandatory Access Control, and Role Based Access Control etc.

Discretionary Access Control (DAC) is a means of restricting access to resource based on the identity of user or membership in certain group. DAC allows users to grant or revoke access privileges to any of the objects under their control. In certain sense, users are said to be the owner of the objects under their control. However, in many organizations, the end users do not own the information to which they are allowed to access. The access priorities are controlled by the organization and are often based on the employee functions rather than data ownership. Mandatory Access Control (MAC) is a means of restricting access to objects based on the sensitivity of the information contained in the objects. MAC secure information by assigning sensitivity labels on information and comparing this to the level of sensitivity a user is operating at. The most important feature of MAC is that the user can not fully control the access to resources that they create. The constraints of the access rules of MAC are provided by security policies which are in the hands of the system administrator. In general, MAC mechanisms are more secure than DAC, but they have trade offs in performance and convenience to users.

Role Based Access Control (RBAC) [82], firstly proposed by D.Ferradiolo etc.,

is a more effective, natural and promising method for controlling access to computer resources, in which access decision are based on the roles that individual users have as part of an organization. Under the RBAC framework, users are granted membership into roles based on their competencies and responsibilities in the organization. The operations that a user is permitted to perform are based on the user's role. User membership into roles can be revoked easily and established with job assignments. This simplifies the administration and management of privileges because roles can be updated without updating the privileges for every user on an individual basis.

Under RBAC, roles can have overlapping responsibilities and privileges. That means, users belonging to different roles may need to perform common operations. Some general operations may even be performed by all employees. Role hierarchies can be established to avoid repeatedly specifying these general operations for each role. A role hierarchy defines role that have unique attributes and may contain other roles.

A properly-administrated RBAC system enables users to carry out a broad range of authorized operations, and provides great flexibility and breadth of application. System administrators can control access at a level of abstraction that is natural to the way that enterprises typically conduct business. This is achieved by statically and dynamically regulating user's actions through the establishment and definition of roles, role hierarchies, relationships, and constraints.

Design of Access Controller

Access Controller is a processing center to make the decision for permission assignment to specific user for specific objects. Different from other access control engine, here the tasks of authentication and authorization are assumed to be done out of component Access Controller. An XML based policy is given as access rules to specify who has what kind of access privilege to which target object and under what conditions. Access Controller makes decision to give the permission on the basis of access policy and received request with roles of current user. A full example of access policy can be referred in Appendix C. The syntax of policy is borrowed from PERMIS project [20] and comprised of the following sub-policies:

- Subject Policy: specifies the domains of users that are allowed to access resource within the overall policy. Each domain is specified as an LDAP sub-tree, using Include and Exclude statements, with optional layering.

- *SOA Policy*: lists the LDAP DNs of the SOAs that are trusted to issue roles to the subjects specified in the subject policy. These DNs will match the root issuer names in published Attribute Certificates stored in LDAP server.

- *Role Hierarchy Policy*: defines the role hierarchies that are supported in this access policy. Each role hierarchy is specified as a set of Superior-Subordinates attribute values. The superior roles inherit the privileges of a subordinate role. The role hierarchy also supports multiple inheritances whereby a superior role inherits all the privileges of a set of subordinate roles.

- *Role Assignment policy*: specifies which roles are allocated to which subjects and by which SOAs. For each assignment, there are also time constrains to specify the valid lifetime of roles assignment.

- *Target Policy*: specifies the target domains covered by this policy. Target domains are specified as LDAP sub-trees using Include and Exclude etc.

- *Action Policy*: specifies the actions supported by this policy. An action, the smallest granularity of the access to a target, has a name, and zero or more arguments. Actions are separated from targets because several targets may support the same action.

- *Target Access Policy*: specifies which roles have permission to perform which actions on which targets and under which conditions. The target access policy comprises a set of target access clauses. Each target access clause grants an initiator with a specified set of roles permission to carry out the specified actions on the specified list of targets.

Component of Access Controller takes target and operation of request and roles of current user as its inputs, and produce a Boolean value to indicate whether the permission is allowed for given user with specified target and operation. The implementation of access enforcement is illustrated as in Figure 5.12. During the initialization of component, the policy will be retrieved and decoded by policy parser which provides rich set of APIs for Access Decision Function (ADF) to extract relevant information from policy. After receiving the request, ADF use the information from request, such as target, operation roles, and information from policy, such as role hierarchies etc., to create an accessing credential used for subsequent checks. First ADF checks whether the credential satisfies the assignment rules. After getting a result of True, ADF continue checks the access rules using created credential. The result of this step will be sent out as the output of permission.

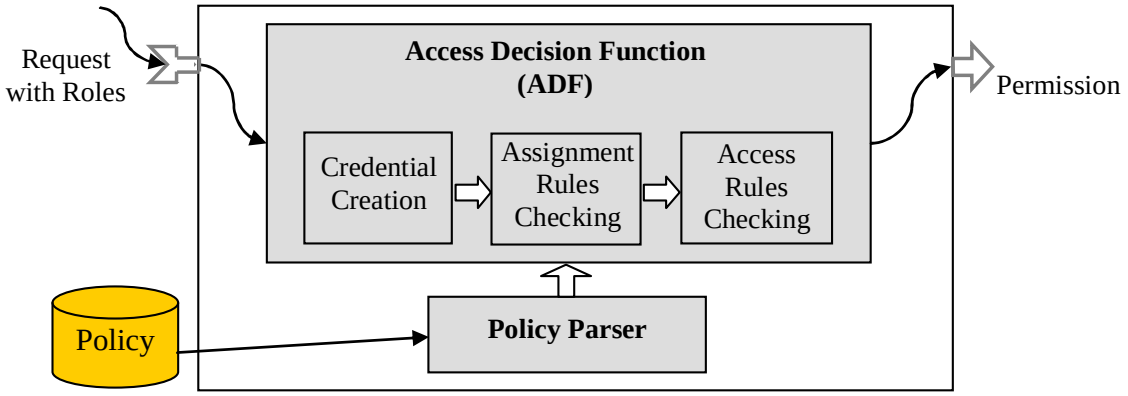

Figure 5.12: Access Enforcement of Access Controller

5.3.5 Security Centre - Local Security Authority

In the previously introduced secure components, Public Key Certificate (PKC) and Attribute Certificate (AC) are used for authentication and authorization. As a aid tool which is indispensable for running SDS3, a Security Center is created to take charge of the administration functions: Graphics User Interface (GUI) based administration for open LDAP server and management for PKCs and ACs. A windows version OpenLDAP built by ILEX team [85] is employed in SDS3 to store PKCs and ACs. The PKC is directly supported in OpenLDAP server. However, the supporting for storage of AC has to manually add an additional item in the schema configuration file, inetorgperson.schema, as follows:

```
attributetype ( 2.5.4.76
        NAME 'AttributeCertificate'
        DESC 'RFC2256: X.509 AC certificate, use ;binary'
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.8 )
```

To conveniently manage LDAP server, two GUI based LDAP functions are provided: creating LDAP entry and removing LDAP entry. The PKCs and ACs will automatically be stored in LDAP server after they are created.

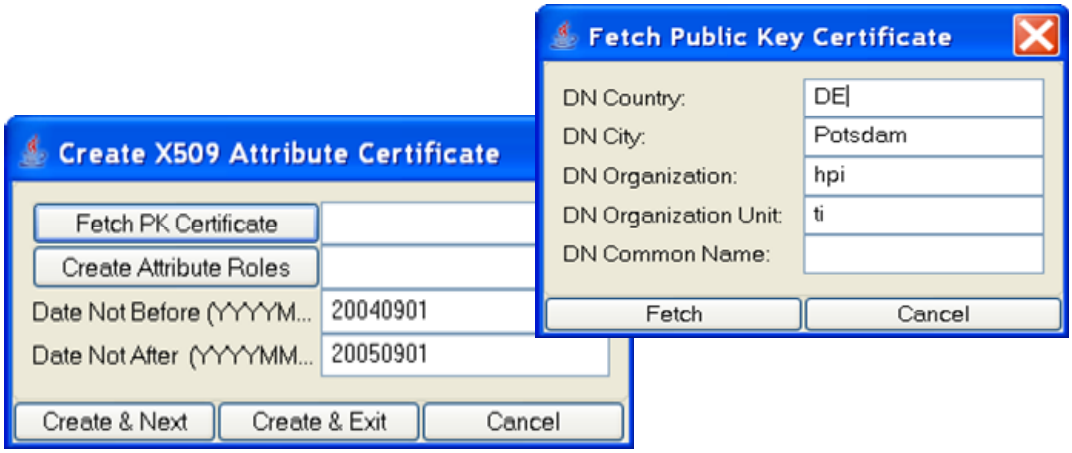

Figure 5.13: Screenshot of Attribute Certificate Creating

In normal case the public key Certificate Authority (CA) and Attribute Authority (AA) are not necessary to be the same entity, but here they are same and managed by Security Center which takes the role of local security authority for both PKC and AC. The implementation of Security Center is based on public research security package - IAIK [46] that provides implementations for all kinds of concrete algorithms, such as algorithm of encryption and digital signature etc. The basic functions offered by local security authority are: creating and revoking Public Key Certificate and Attribute Certificate. Figure 5.13 is a screenshot for attribute certificate creating. Other trusted CAs and AAs can also be configured into trust list to enable the SDS3 accepting the PKCs and ACs issued by other CAs and AAs.

5.4 Analysis on Features and Applications

As a middleware system, SDS3 is able to ease the burden of programmer to develop distributed applications. Apart from this basic functionality, SDS3 is distinguished from other middleware solutions in two important aspects: security and dynamics. Instead of behaving as middleware services, security functionalities in SDS3 are integrated into the core components of middleware system to establish an authenticated and authorized Role Based Access Control (RBAC) framework to enhance the security on invocation of applications. In addition, the technology of Routing Based Workflow (RBW) is employed to manage the secure components, which results in high flexible and dynamic management of security control. The detailed features of SDS3 can be classified into the following different aspects:

Multi-Level Security Control

The security controls of application invocations in SDS3 are realized in two levels: the coarse-level strategy control and detail-level control on specific application. Coarse-level security control means the specific application-independent control strategy. For instance, if all the accessing users are assumed to have the real identity that he claims, and then they don't need to be authenticated. The coarse-level security control is realized by the organization of different secure components: Authenticator component for user authentication, Authorizer component for user authorization and Access Controller component for access enforcement function. The detail-level security control is concerned about secure accessing on the specific application or even the concrete one or more operations of specific application. For example, the accessing to company's sensitive data, e.g. developing blueprint, is only allowed for the entity with role of Manager. The detail-level security control is realized by policy configuration for component of Access Controller.

Dynamic changeable Coarse-Level Security Control

Due to the RBW management on secure components, the coarse-level security control of SDS3 can be dynamically changed during runtime. In SDS3 changing the collaboration relation of secure components means a new security control strategy is generated. In another words, a routing means a solution for one security control strategy. If all introduced components are utilized in one routing, then the full and powerful security are provided: each access will be authenticated, authorized and control policy checked. However, one or more secure components may also be omitted to provide incomplete secure invocation for less important applications. When one component is missing for routing, the output values for this component will be replaced with corresponding default values that are configured in XML based configuration language. For example, if all accessing users are assumed to be the real entity that it claims, the component of Authenticator can be omitted. The collaborative component, such as Authorizer, will automatically get a "True" as the default value from component of Authenticator. Likewise, if the applications are given the same access

permission for all users, the component of Authorizer is not necessary and should be omitted in the routing. The simplest routing just contains one component, Request Broker. In this case there is no any security control on the application invocation.

Secure Application without Security Source Code

Applications of the SDS3 contain only business processing logics, and application developer does not need to consider any security issues in the source code. All the security issues of application can be configured in policy after application is programmed. Even the application specific security control, such as access permission to an operation, can also be configured in the policy for component of Access Controller. As illustrated in Fig.5.14, the example shows that which roles (e.g. "Designer") are allowed to access which operations (e.g. "inverse") of the specified application (e.g. "AppSample"). A full policy example can be referred in appendix C.

```
- <TargetAccessPolicy>
  - <TargetAccess>
    - <RoleList>
        <Role Type="UserRole" Value="Manager" />
        <Role Type="AuthorRole" Value="Designer" />
      </RoleList>
    - <TargetList>
      - <Target Actions="inverse">
          <TargetDomain ID="AppSample" />
        </Target>
      </TargetList>
  </TargetAccess>
  ...
</TargetAccessPolicy>
 ...
```

Figure 5.14: Example of Sub-Policy of Target Access Policy

Supporting Multiple Security Control Strategies

Multiple security control strategies are supported in SDS3 to enable flexible security control for different kinds of applications. To be specific, in an application server powered by SDS3, it is possible to support two or more different control strategies: sensitive applications need strict and full control, less important applications need only incomplete security control, and testing applications do not need any security control. This feature is directly inherited from the RBW which supports multiple routing synchronously. In RBW, each component instance does not reside in one routing for long time. After execution, instance will be unloaded from routing. For component instance, it has no awareness that there is only one routing or multiple routings. There is no differences that component instance is executed in the same or different routings at different time. So multiple routing can be created to share the same instances of components to support multiple solutions synchronously for different requirements.

Chapter 6

Case Study: Dynamic Services Composer

Web services composition is a promising approach to fulfill the enterprise application integration with which enterprise applications are able to interact in the application level regardless of their deployment location, implementation language, and the supporting fundamental technology of each single application. This chapter presents another application case of Routing Based Workflow (RBW) - Dynamic Services Composer (DSC), which is capable to integrate distributed web services and enable the composition structure dynamically changed. In DSC, the technology of RBW is used to model the interaction and process of collaborative web services, and the open source project Apache Axis and Web Services Invocation Framework (WSIF) are employed to help constructing the system.

6.1 Introduction to Services Composition

Web services composition is an emerging paradigm for applications integration within and across organizational boundaries. Numerous approaches and techniques for web services composition has continuously been proposed from different vendors and coalitions. Most approaches for web services composition can be classified into two categories: process oriented composition and semantic based composition.

6.1.1 Process Oriented Composition

The process oriented composition approaches are mostly supported and pushed forward by industrial enterprises. Early works included XLANG [87], WSFL [54] and WSCL [6] etc. The XLANG specification was developed by Microsoft alone for the Microsoft BizTalk Server. The XLANG focuses on the creation of business processes and interaction between web service providers. The Web Services Flow Language was proposed by IBM to define a specific order of activities and data exchanges for a particular process. Two kinds of models, i.e. flow model and global model, are defined in WSFL. The flow model represents series of activities in the process, while

the global model binds each activity to a specific web service instance. The WSCL is the abbreviation of Web Services Conversation Language proposed by HP. The WSCL outlines a rather simple conversation language standard, and focuses on modeling the sequence of interaction messages between web services. Two rather new specifications, BPEL4WS and WSCI, are introduced as follows.

BPEL4WS

The specification of BPEL4WS [3] was proposed by industrial Microsoft, IBM, Siebel and SAP etc. The BPEL4WS, based on early works of XLANG and WSFL, models the behavior of web services in a business process interaction. BPEL4WS supports for executable and abstract business processes. An executable process models the behaviors of participants in a specific business interaction, essentially modeling a private workflow, illustrated as in Figure 6.1. Abstract processes specify the public message exchanges between parties, but it is not executable and also does not convey the internal details of a process flow. The specification includes both basic and structured activities. A basic activity can be taken as a component that interacts with external services and process itself. The constructed activities manage the overall process flow, specifying what activities should run and in what order. Containers and partners are two other important elements in BPEL4WS. A container identifies the specific data exchanged in a message flow, and are used to manage the persistence of data across web services requests. A partner could be any service that the process invokes or any service that invokes the process.

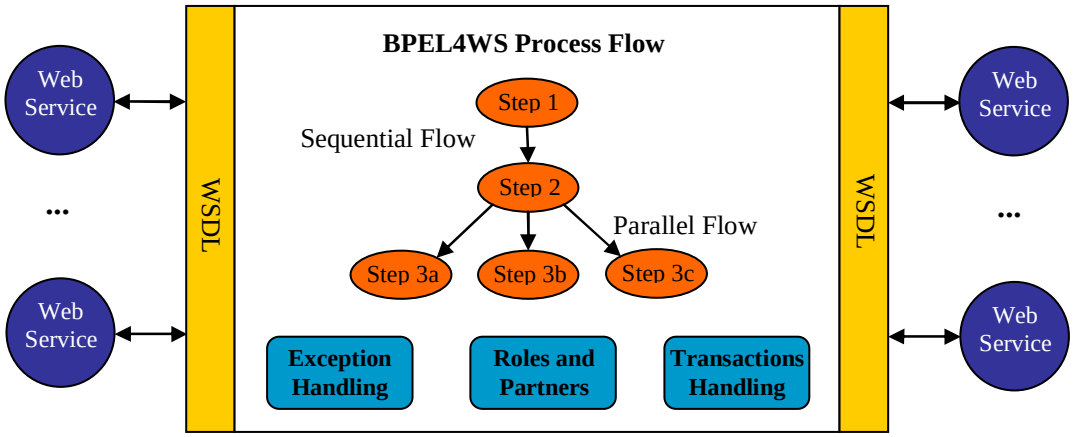

Figure 6.1: Working Diagram of BPEL4WS

WSCI

The Web Services Choreography Interface (WSCI) [4] is a specification proposed by Sun, SAP, BEA and Intalio etc. The WSCI defines the overall choreography of web services that describes the messages between services that participate in a collaborative exchange. A key aspect of WSCI is that it only describes the observable or visible

behavior between web services. WSCI does not address the definition of executable business process as defined in BPEL4WS. Furthermore, a single WSCI document only describes one partner's participation in a message exchange. As illustrated in Figure 6.2, WSCI choreography includes a set of WSCI documents, an each one is for a partner in the interaction. In WSCI, there is no single controlling process managing the interaction. WSCI supports also basic and structured activities which is for sequential and parallel process etc.

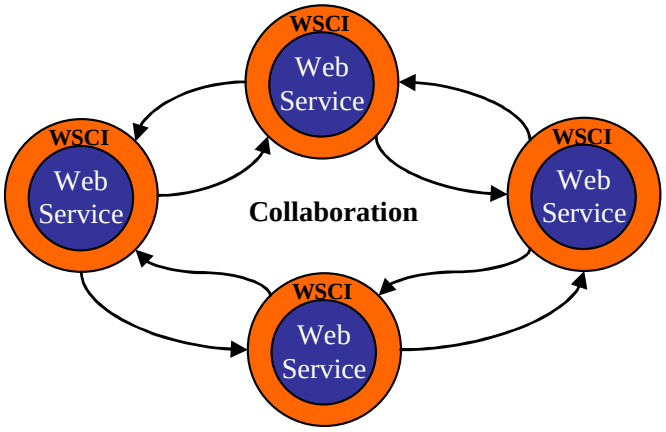

Figure 6.2: Working Diagram of WSCI

6.1.2 Semantic Based Composition

The semantic web community concentrates on reasoning about resources by explicitly declaring their preconditions and effects with terms precisely defined in Resource Description Framework (RDF) [49]. OWL-S (formerly DAML-S) [58] is a semantic markup for web services, which describes the properties, capabilities, interaction and access model of web services in an unambiguous, computer-interpretable form. The tasks that OWL-S is expected to address are automatic web service discovery, invocation, composition and interoperation. The ontology of services in OWL-S consists of the following three main parts:

- *Service Profile* provides the information to tell what the service does, in a way that is suitable for a service-seeking agent to determine whether the service meets its needs.

- *Service Model* tells a client how to use the service by detailing the semantic content of requests, conditions under which particular outcomes will occur, and step by step processes leading to those outcomes.

- *Service Grounding* specifies details of how an agent can access a service. Typically a grounding will specify a communication protocol, message formats, and other service-specific details, such as port numbers etc.

6.2 Modeling - RBW for Services Composition

Routing Based Workflow (RBW) is originally proposed to model and manage the collaborative software component and applied in middleware construction introduced in Chapter 5. Here the RBW is employed again to model the process and interaction of web services and achieve the capability of dynamic change on the services flow structure. The RBW modeling for web services composition consists of the modeling of services activity, services process, interaction and control etc., which are introduced in detail as follows.

6.2.1 Basic Service Modeling

Basic web service is an independent and atomic web service activity executed in services process. A basic web service may come from internal or external, and may be a real atomic web service or even a composite service that behaves as an atomic service. In RBW for services composition, a basic service is modeled as service component, which evolves from the component structure of original RBW.

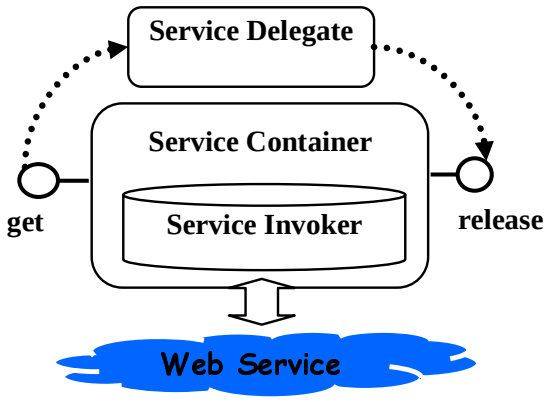

Figure 6.3: Diagram of Service Component

As depicted in Figure 6.3, service component is comprised of service delegate, service container, service processor, and the remote web service entity. The entity that directly invokes web service is service invoker. Service container takes charge of the life cycle activities of a service invoker. During a process, each service component may create one or several instances of service invoker which are all hosted in service container. Service delegate is responsible for management and control tasks of service component during process, and service container is the bridge between service delegate and service processor. Just the same as component in RBW, service invoker, the implementation of service invocation, is also temporarily bound to the service delegate. To be more specific, only when web service is demanded to invoke, the service delegate will ask service container to assign an instance of service invoker to perform the invocation operation. After service invoker finishes its invocation, service delegate will release this instance and send it back to container.

In term of implementation, service delegate and service container are exactly the same as component delegate and component container introduced in Chapter 4. For the sake of better understanding in new application of web services composition, the names are changed here. Service invoker is actually an example of function-specific component processor, just as component of Authenticator in SDS3. Service invoker also inherits basic functionality from generic processor which is responsible for contact keeping and communication with service delegate and service container. The functional component part of service invoker fulfils the implementation of invocation to a specific web service. The communications among different service delegates and between service delegate and its corresponding service invoker are realized by communication ports introduced already in Chapter 4.

6.2.2 Flow Control Modeling

Control flow modeling specifies execution order and data flow direction of involved basic web services through the communication ports and a set of control links of RBW. The sequential and parallel flows can be realized by the binding pairs of communication ports. The control links fulfil the conditional logic to enable different kinds of conditional flow in which one component will be picked from multiple candidates for next execution . As introduced before, the control links include three logic links, i.e. AndLink, OrLink and XorLink, and one data map link, i.e. MapLink. Different kinds of control flows realized in RBW are explained as follows:

- *Sequential Flow*: Service component starts to execute only after its in ports (namely, target ports of the binding pairs) receive the data from out ports (namely, source ports of the binding pairs) of other services. Each service component may have multiple in ports which may be bound to out ports belonging to different service components. Only after all of in ports receive the data, the service component will then be triggered to execute.

- *Parallel Flow*: is directly supported through communication ports. When several out ports of service A is bound to multiple in ports that belong to different service components, the data from service A will then flow to different followed services in parallel way.

- *Cycled Flow*: is also directly supported, similar to parallel flow. If one out port of service A is bound to in port that belongs to the same service or a former executed service, then cycled flow is formed.

- *Logic Control Flow*: is conditional flow in which the decision is made according to the input values and logic operation of AND, OR, or XOR. These simple logic computations are realized by three control links: AndLink, OrLink, and XorLink. These three control links include multiple in ports and two out ports. The ports of control link are designed for specific data with type of Boolean and will also be bound to ports of service component.

During the composition of basic web services, all above mentioned flow models will be employed and mixed together to form a complex flow model. For the usage of control link is also realized by port binding with ports from service components, all kind of flow models are designed by the layout of communication ports.

6.2.3 Composite Service Modeling

Composite service models process and interactions of basic web services. To be specific, composite service specifies how the involved services are combined, in which order they have to be performed, and how the data are transferred among involved service activities. In RBW for service composition, the composite service is modeled by routing in which service delegates and control links construct the primary parts, and the binding pairs are created to joins isolated service activities together to form a connected services flow. As illustrated in Figure 6.4, the composite service is composed of basic services, and itself can also behave as a basic service that is possible to be integrated in other composite services. When it is accessed, client does not know whether it is an atomic service or a composite service.

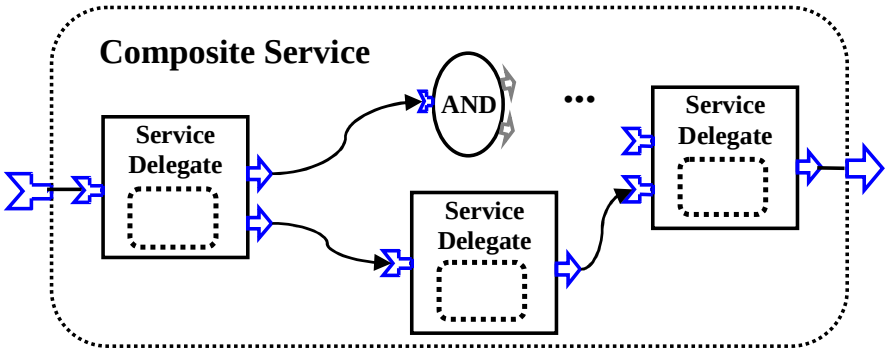

Figure 6.4: Diagram of Composite Service

In term of implementation, a composite service is just a routing, so the management and execution of composite service is exactly the same as management and execution of routing. The execution entities, i.e. instances of service invoker, are stored in repositories. The execution of a composite service is a procedure to temporarily load the invoker instance, trigger the execution of service invoker, and unload the invoker instance. The distinctive structure of basic service and composite service leads to the temporary binding for instances of service invoker, which enables dynamic changes on the structure of composite service. The execution and change of composite service need series of control activities to coordinate and keep consistency for all involved basic services. Based on the data flow dependencies created from the information of ports binding pairs, control dependencies are established for all relevant services. Control events are designed to carry the information and commands of control activity, and the distribution of control events is realized through the established control dependencies. Details for execution and management of composite service (namely routing) can be referred in section 4.3.

6.3 System - Dynamic Services Composer

Based on the RBW modeling for service composition, we developed the Dynamic Service Composer (DSC), which is able to integrate the single internal or external web services to a more powerful composite service and to dynamically change the composite service during running time. In DSC, two open source project are employed: Apache Axis [32] which is composed as the underlying infrastructure to publish self-developed web services and integrated composite services (DSC services), and Apache Web Service Invocation Framework (WSIF) [33] which is used to construct Service Invoker to implement the invocation to extern web services. Figure 6.5 illustrates the architecture of DSC and shows an overview that how services are composed, executed and published in our system. The Key technologies involved in DSC are introduced respectively in the following subsections.

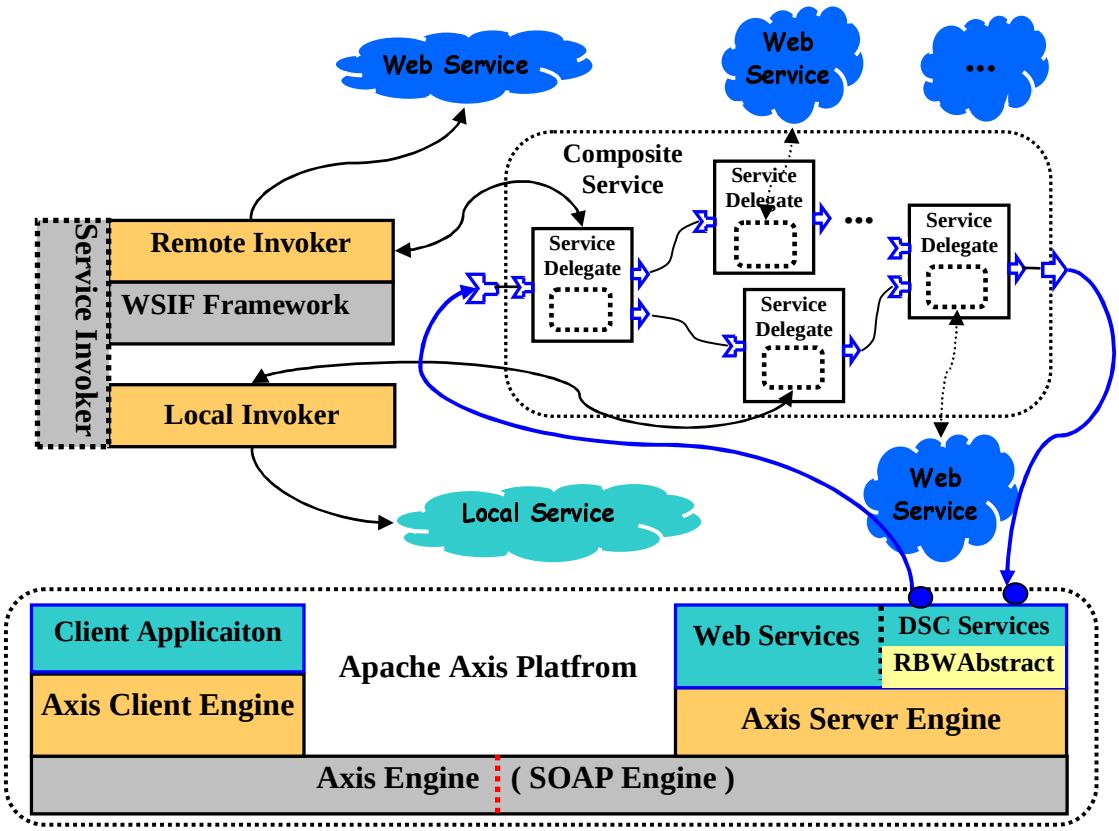

Figure 6.5: Architecture of Dynamic Services Composer

6.3.1 RBW Integration

As depicted in Figure 6.5, the open source project Apache Axis acts as the underlying platform of DSC to publish services, and enable the services to be accessible. A module of RBW Abstract is designed to integrate RBW with Axis, similar to Workflow Adapter of SDS3, to publish and execute composite services. With RBW abstract

the execution differences between composite services and other atomic services are transparent to clients. To enable the composite services to be accessed in a way the same as that of other atomic web services, the IO behaviors, e.g. parameters and return type, of composite service have to be created using the same way as that for atomic service creating which is provided by Apache Axis. The implementation of composite service is the codes which construct and execute RBW request based on RBW Abstract. The RBW Abstract provides a set of APIs to support the integration of RBW with Axis. As illustrated in Figure 6.6, the functionality of RBW Abstract primarily focuses on the following aspects:

- *Constructing RBW Request* based on the IO behaviors of composite service. The primary tasks are to create named objects as input set of composite service with the given parameter information, and to create named objects as output set with the given information of return type.
- *Pre-Execution of RBW*: the actual execution is managed by RBW manager. Here only some preparations of execution are done, such as assigning a routing according to the name of composite service etc.
- *Extracting Result from RBW Response*: After execution the result is extracted and returned as the execution value of composite service.

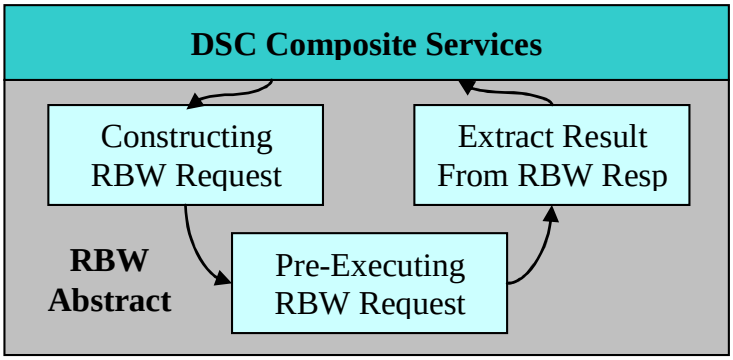

Figure 6.6: Diagram of RBW Abstract

6.3.2 Services Invocation

The implementation of invoking all kinds of remote web services is not an easy task, because web services may be deployed in heterogeneous environment using varied supporting technologies. The implementation of Service Invoker in DSC has employed an open source project Apache Web Service Invocation Framework (WSIF) [33], which provide a uniform means to access web services described in WSDL, no matter how and where the services are provided. WSIF tries to free the developer from constrains of developing services for particular transports or service environments. Thus, a set of APIs are provided in WSIF to allow accessing any web services whose transport

bindings are supported by SOAP, Java RMI or HTTP Get etc. WSLF supports traditional approach of stub mechanism or completely dynamic invocation of web service based upon examination of the meta-data about service at runtime. WSIF invokes service operations through the following steps:

1. Load a WSDL document that describes the invoked web service.
2. Create a port factory for this service and retrieve a specific service port with the given name of port type.
3. Create input message and output message, if necessary, by using message parts typed according to some native type system
4. Make the invocation by supplying the port with the name of the operation that is going to be invoked, along with an input and /or output message.

In implementation, the Service Invoker inherits from Generic Processor which is an abstract class for each component managed by RBW, and provides concrete implementation for service invocation. The web service accessed by Service Invoker may be a service deployed locally or a service deployed remotely. Although the local service can be dealt with the same way as that for remote web service, we create a Local Invoker to separate itself from Remote Invoker to improve the efficiency and speed the invocation. For Local Invoker, it does not need the document of WSDL and requires direct invocation information, such as name of service, port type, operation, and parameter names and values etc. For Remote Invoker the WSDL of diverse services has to be explicitly specified. During the execution all instances of different service components are instantiated from the same class of component entities, such as service delegate, service container, and service invoker etc. The configuration information, e.g. parameters and in/out ports etc., are used to initialize and customize an instance of the service component exclusively for a specific web service.

6.3.3 Services Configuration

The XML based configuration language for RBW can also be used to specify how to compose a complex service from multiple basic services. The key configuration elements are the basic service activity and the composite service activity introduced as follows. The configuration of control link is similar to that of basic service activity and a full configuration example is given in Appendix D.

Basic Service Activity

For each web service that needs to be integrated into a composite service, it has to be firstly configured as a basic service activity using the XML based configuration language of RBW. Each basic service activity is specified according to one operation provided by a web service. If two or more operations of a service need to be integrated into composite service, those operations have to be specified respectively into different

basic service activities. The configuration of the basic service activity specifies the information of service invoker, parameters, properties and communication ports etc., as shown in Figure 6.7. The configuration of basic service activity has to be complied with the component schema introduced in section 4.5. The key information which are used to customize service component, are provided by series of parameters. Three parameters are used to identify the function of remote web service invoked by this service component:

- *WSDLDoc* indicates the WSDL document of the target web service;
- *Interface* indicates the target interface (port type) of the specified web service;
- *Operation* indicates the target function of the specified web service interface.

In addition, the port information will be used to create service delegate to enable management by RBW.

```
<BasicService = "CurrencyConvertor" classType ="ti.ws.handler.RemoteInvoker">
   <parameter name="WSDLDoc" classType="xsd:string>.\wsdl\currency.wsdl</parameter>
   <parameter name="Interface" classType="xsd:string>ConversionRateSoap </parameter>
   <parameter name="Operation" classType="xsd:string>ConversionRate </parameter>
   <property name="PoolSize"> 5 </property>
   ...
   <port identifier="CurrencyConvertor#fromCurrency" classType="xsd:string">
      <inDirection> in </inDirection>
      <portType> operation </portType>
      <usage> required </usage>
   </port>
   ...
</BasicService>
```

Figure 6.7: Simplified Configuration for Basic Service Activity

Composite Service Activity

The configuration of a composite service specifies which basic services will be involved to the composition and how these basic services interact to each other. The configuration of composite services has to comply with the schema of routing introduced in section 4.5. As shown in Figure 6.8, the configuration comprise of four parts:

- *inputs* and *outputs* specify the IO behaviors of a composite service;
- *import* specifies which basic web services are integrated into this composite service;
- *controlLink* specifies which control links are used in this composite service;
- *connectors* specifies binding pairs of communication ports which are used to schedule how the data are flowed among basic services or control links.

```
<CompositeService = "CurrencyConvertor">
  <inputs>
    <namedValue identifier="Request#fromCurrency" classType="xsd:string"/>
    <namedValue identifier="Request#fromCurrency" classType="xsd:string"/>
    ...
  </inputs>
  <outputs>
    <namedValue identifier="Response#return" classType="xsd:double"/>
    ...
  </outputs>
  <import>CurrencyConvertor</import>
  <import>EmailVerifer</import>
  <import>EmailSender</import>
  ...
  <connectors>
    <link>
      <source>Request#fromCurrency </source>
      <source>CurrencyConvertor#toCurrency</source></link>
    ...
  </connectors>
</CompositeService>
```

Figure 6.8: Simplified Configuration for Composite Service Activity

6.3.4 Services Deployment

The deploying and publishing of a composite service in the DSC are closely related to the Apache Axis which is integrated as the underlying platform of the DSC. As the third generation of Apache SOAP, Axis is not only a SOAP engine, but also a simple stand-alone server to allow easily accessing remote services and deploying self-developed services. In DSC, the composite services are marked as DSC Service. In fact the way to deploy a DSC Services is similar to that of a basic service, which can be accomplished by adding a service entry in the Web Service Deployment Document (WSDD) of Axis Server. The different point is the implementation of service. For a basic service, the implementation is the code to realize the claimed functionality. For a DSC Service, the implementation is the code to execute the composite service. RBW Abstract is a package that provides foundational APIs to execute composite services, such as initializing the RBW, creating RBW request etc. A typical implementation of a composite service is illustrated as the following Figure 6.9.

```
Public boolean query_info( fromCurrency, toCurrency, subject, emailAddress ){
      operationName = "query_info";
      SOAPServiceRequest request = new SOAPServiceRequest(serviceName, operationName);
      String[] paraTypes = new String[] {};
       Object[] paraObjects = new Object[]{fromCurrency, toCurrency, subject, emailAddress};
      String[] returnTypes = new String[]{};
      success = construct_request (request, paraTypes, paraObjects, returnTypes);
      If(success) { execute_composite_service(request);}
      Object result = request.get_response().get_return_value();
      return result;
}
```

Figure 6.9: Pseudo Code of a Typical Implementation of Composite Services

The key steps to deploy a new composite service can be summarized as follows:

1. Specify basic service activities and composite services using XML based configuration language.

2. Construct an interface of composite service as one function of an application in the Apache Axis, and then, implement the function as in Figure 6.9.

3. Add a service entry in the WSDD configuration of the Axis server to enable the DSC application accessible by clients.

6.3.5 A Practical Example

During the procedure of web services composition, local services may be used in two cases: i) a local services is required to realize the self-developed functional service; ii) a local service is required to deal with coordination among different services. The services coordination may be information integration, type conversion or even simple logical computing. A practical example is introduced here to provide a composite service of querying stock price and getting result information by email, as depicted in Figure 6.10. The composite service has integrated four services: Stock Query which retrieves stock price information with the given stock, Email Verifier which outputs a boolean value to indicate whether the given email is valid or not, Email Sending which send an email according to the given information, and Decision Maker which is a local service to convert query result into the email message body and judge whether it is necessary to email the query result. The XMethod website[1] provides a portal to publish free web services, from which above three services are found.

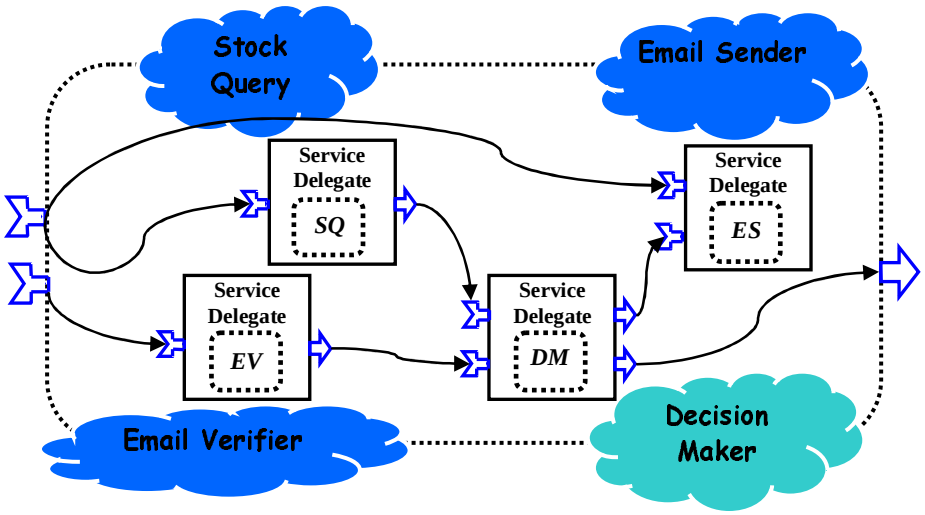

Figure 6.10: The Composite Service of Stock Query and Sending

[1]http://www.xmethod.net/

6.4 Discussion and Analysis

So far the Dynamic Services Composer (DSC) has been introduced from its modeling with RBW to the system architecture. The most important advantage of DSC is the capability of dynamic change on the service providers and flow structure of composite services, which is described as follows.

Dynamic Change on Service Composition

The dynamic capability of DSC is directly inherited from RBW which is specially designed for dynamic changes. The dynamic binding of service provider, supported in most approaches for dynamic service composition, is also supported in DSC by changing the service invoker or changing the whole service component. The more attractive capability of dynamism in DSC is to change the flow structure of composite services during running time. In DSC a composite service is represented by a routing which models the execution environment for collaborative services. So all the change capabilities on the composition structure, e.g. adding/removing/replacing a web service and reorganizing the interaction way of collaborative services etc., can be achieved through changing a routing, which is already introduced in section 4.4

Actually our approach of RBW is centralized on the execution management of components or web services, not the specification definition which is the target of the process oriented composition approaches, e.g. BPEL4WS, and semantic based composition approaches, such as OWL-S. It would be better to combine the BPEL4WS or OWL-S into the RBW to provide more powerful functions to automatically and dynamically integrate web services.

Shortages for Enterprise Application

We have to confess that the DSC is still not mature. The key contributions of DSC is that it demonstrates the feasibility of RBW in applications for web service composition. For utilization in enterprise application, numerous works are still demanded to improve the system, such as:

- *Transaction*: atomic behavior of composite service is essential for enterprise application. To keep consistence in the whole distributed system, all executions of involved basic services have to be succeeded, otherwise, all are failed.
- *Security*: Security has to be considered when the integrated basic services are from different organization or different companies.
- *Data Format Transformation*: In different countries or different organization, the same data are possible to be expressed in different formats which should be transformed automatically and transparently.
- *Graphic User Interface (GUI) based Tool*: To easily and quickly compose the web services, GUI based composition tools are needed to improve the efficiency and eliminate the errors.

Chapter 7

Performance Tests and Analysis

The prime contribution throughout my work introduced in this dissertation is Routing Based Workflow (RBW) which achieves a high flexibility of dynamic change on component oriented software systems. The dynamic capability of RBW is achieved by its flexible structure and distinctive execution mechanism which may result in prices of performance. This chapter presents an overall performance tests on RBW and SDS3 to make sure how much expenses of performance are paid, and whether the reduced performance will affect the execution of SDS3 applications.

7.1 Performance Tests

The performance tests are made on the RBW for components deployed in the same computer, not on RBW extension for distributed components. The test environment is a Dell desktop PC with Intel CPU 2.5 Ghz, RAM 2.0 G and Windows XP installed. The performance tests focus on three aspects: running times of key stages of RBW execution, running times comparison between sub-stages and between SDS3 and CORBA, and running times of dynamic changes.

7.1.1 Running Times of Execution Stages

To go into deep details of the RBW execution performance, the tests on different sub-stages of RBW execution are made. Eight blank components are specially designed to test the variety and evolution of RBW execution in different situations. The blank components are designed to simulate all non-functional component behaviors during the RBW execution. The construction of blank component is exactly the same as other components, such as component of Authorizer in SDS3. The features shared by all blank components are described as follows:

- No parameter is specified for component instantiation.
- Only one property is specified: *Pool Size* is set the value 1 to indicate that component container holds only one component instance.

- No functional logic is implemented in component processor. All data got from in ports will be immediately forwarded to out ports.

Each blank component distinguishes itself from other blank components in the aspects of number of in/out ports which are closely related with the performance of RBW. Eight blank components, named respectively *Comp_I1_O0*, ..., *Comp_I4_O4*, are designed to have the communication ports respectively with the number from 1 to 8. For example, component *Comp_I2_O3* has two in ports and three out ports. It is clear that the blank components can not simulate the execution of components in which different parameters or properties may bring huge difference in running time. Aforementioned blank component are only used to test the performance of RBW execution which focuses on the management and non-functional execution. The performance tests of three key sub-stages are introduced as follows.

RBW Initialization

The RBW initialization is comprised of two parts: XML based schema parsing and routing instantiation. The performance test of schema parsing is to compute the running time of parsing from XML based schema to language based schema. In the test eight different configurations are created, and each of them has only one routing schema which involves different numbers of blank components. In order to test the relation between schema parsing time and the number of components involved in the routing, increasing number of components are designed in a series of configurations. For example, the second configuration *config2* contains a routing where there are two components. For each configuration, the test is repeated eight times, respectively called *No.1*, ..., *No.8*. The result of eight tests is shown in Figure 7.1, from which it can be concluded that the parsing time is much affected by the number of components of a routing contained in the configuration.

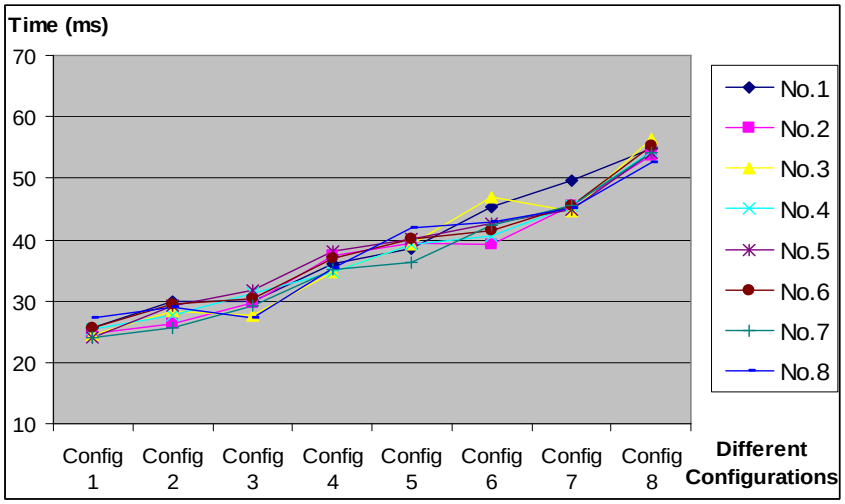

Figure 7.1: Parsing Times for Different XML Configurations

The routing instantiation includes a series of operations, such as instantiating

components, creating component containers and component delegates, and creating routing instances with component instances and input/output information etc. This test is made on the routings that are configured as in Figure 7.1. The test result of instantiation time for different routings is shown in Figure 7.2, from which it can be deduced that although the instantiation time increases with the number of components, the increased amount is very small. Because the instantiation of the first component has to initiate a class loader, and the subsequent components use the same class loader to finish their processing of instantiation.

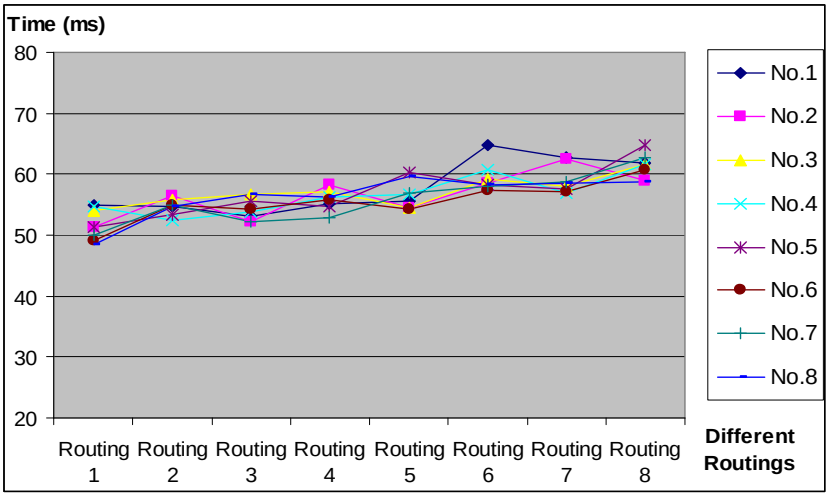

Figure 7.2: Instantiation Times for Different Routings

Virtual Binding

The virtual binding is to connect the instantiated routing. This processing is comprised of connecting inputs/outputs of the routing with ports of corresponding component delegates and connecting different component delegates according to the scheduled binding pairs. The test of virtual binding is made on the instantiated routings introduced in the last section. The test result, shown in Figure 7.3, indicates that the running times of the virtual binding increase with the number of components. When the numbers of components grow to a big value, the running times for virtual bindings increases more quickly. This is because the virtual binding is mainly concerned on the processing of communication ports, such as searching, checking and matching etc. The new added ports will increase not only the time for processing new ports, but also the searching time of all existing ports.

Real Binding and Unbinding

The real binding and unbinding are two important steps for request execution within RBW. The data arrival triggers a real binding to load component instance and to connect the IO ports of component instance with the IO ports of corresponding component delegate. This process occurs in every component for every request execution.

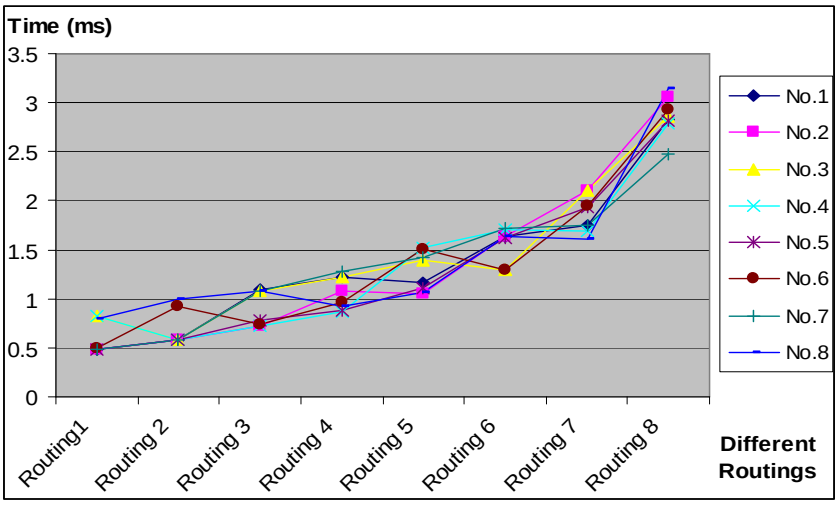

Figure 7.3: Running Times of Virtual Binding for Different Routings

The test was made on the *Routing 8* in which all eight blank components are involved. As shown in Figure 7.4, the processing of real binding consumes a little amount of time although the times increase with the number of IO ports of the component.

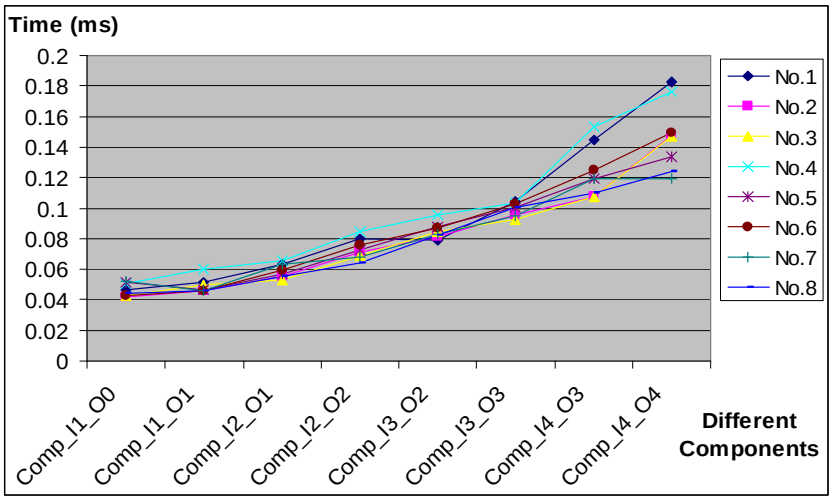

Figure 7.4: Running Times of Real Binding for Different Components

After the component instance finishes its functional execution and sends out the results, the component delegate carries out series operations of real unbinding to unload the component instance and returns it to component container. The unbinding operation occurs for each in/out port of component delegate. After in/out port sends out data from/to component delegate to/from component processor, the in/out port makes an unbinding operation to disconnect itself with corresponding port of component processor. Because this unbinding is dealt in each port, the running times increase quickly with the number of ports, as shown in Figure 7.5. During the request execution, besides real binding and unbinding, there are some other trivial

operations, such as data transferring between different in/out ports and control events distribution etc.

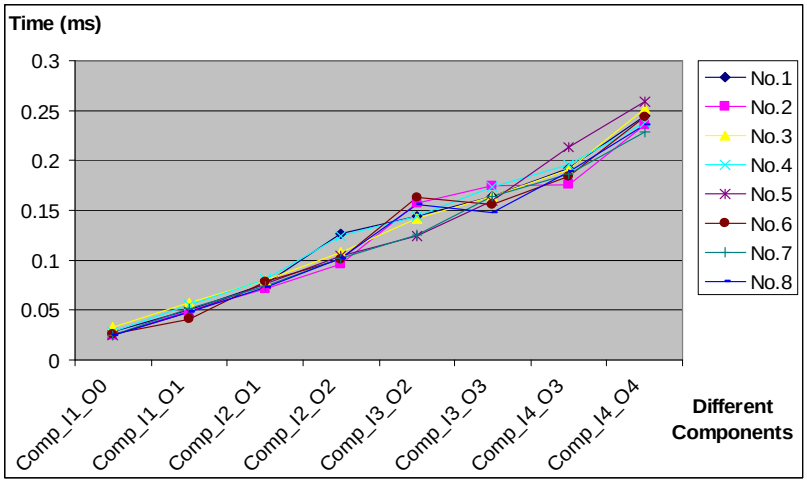

Figure 7.5: Running Times of Real Unbinding for Different Components

7.1.2 Running Times Comparisons

Two kinds of running time comparison tests are introduced in this section: comparison of different stages in one request execution and comparison between CORBA and SDS3 for execution of invocations to the same applications. The blank components are not used for tests here, and the comparison tests are made directly on our middleware system SDS3.

stage comparison inside of Request Execution

The stage comparison test is made to get the running time distribution in the execution of real requests. This test directly shows how much performance expenses are paid for the advantage of dynamic change when employing RBW. During the rest, only the important stages are recorded and other trivial operations are missed here. The stage of routing schema parsing, routing instantiation and virtual binding are executed when the RBW initiates, so they are tested by repeated starting up and shutting down of the SDS3. The real binding/unbinding and execution of components are tested by repeated request invocations. As shown in Figure 7.6, all results are statistical average of eight repeated tests. From the comparison figure, most non-functional running times are consumed during the initialization stages, such as parsing, instantiation etc. During request executions, the key non-functional operations are real binding and unbinding whose running times (average level less than 0.1 ms) are much less than that of RBW initializations and functional executions.

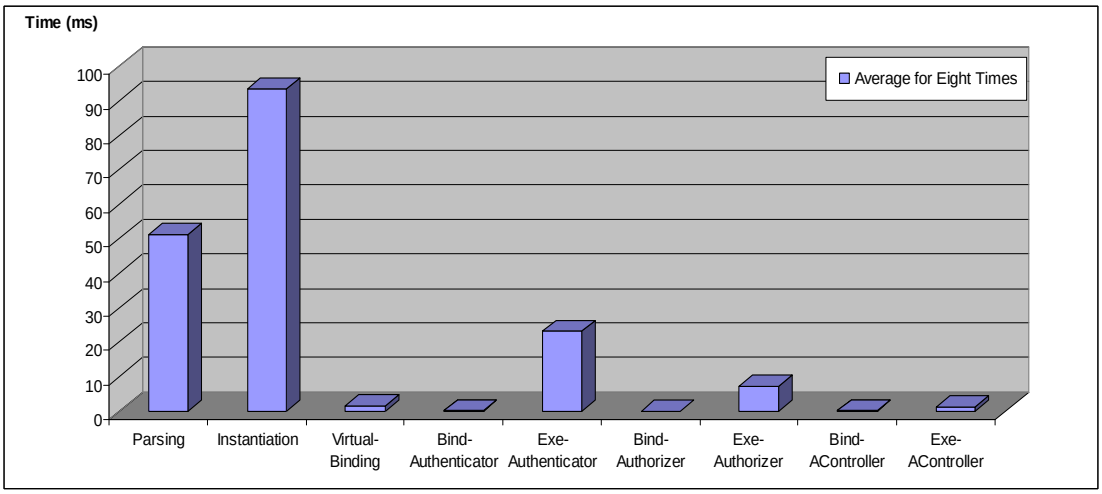

Figure 7.6: Running Time Comparison of Different Execution Stages

Request Execution comparison between SDS3 and CORBA

Because the SDS3 is built on top of Object Request Broker (ORB) of CORBA, a performance comparison test between SDS3 and CORBA is made to evaluate the efficiency of SDS3. In the test, the client program and server program are deployed in the same environment introduced in the beginning of this chapter. To be comparable with SDS3, the tested CORBA are configured with modules of DII, DSI and POA. The SDS3 are configured with three supported solutions:

- *SDS3-NoSecurity*: there is only one component, Request Broker, to deal with the invocation on its applications.
- *SDS3-HalfSecurity*: All the access users are automatically considered as the person who claimed. So, three components are used to deal with requests: Authorizer, Access Controller, and Request Broker.
- *SDS3-FullSecurity*: A full and strictest security checking is made on the application invocation through this solution, in which four components are employed: Authenticator, Authorizer, Access Controller, and Request Broker.

In the test, all requests are created to invoke a same application deployed respectively in CORBA and SDS3: an application to realize the arithmetic operation of *divide*. All requests are repeatedly executed ten times and the results are shown in Figure 7.7. From the test result, the invocation by solution of SDS3-NoSecurity is a little faster than invocation by CORBA. This is because the module of Workflow Adapter is designed more simply than POA. To adapt for enterprise applications in the future, the Workflow Adapter need to be improved to be more powerful, and will cost a little more time. The request invocations by SDS3-HalfSecurity and SDS3-FullSecurity certainly cost a little more time than CORBA because they employ more components for security checking.

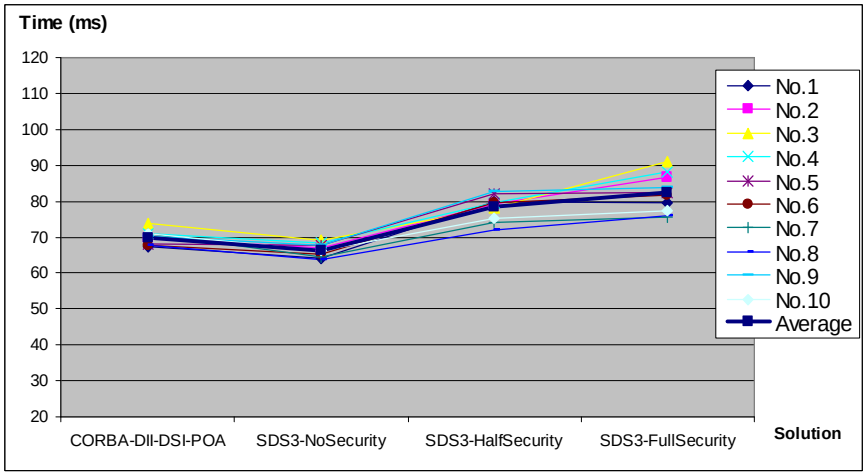

Figure 7.7: Running Time Comparison between Different Solutions

7.1.3 Running Times for Dynamic Changes

Dynamic changes on the components structure are the most important contribution of RBW. In RBW, the routing level changes, such as removing or cloning a routing solution etc., are rather easy and not explained here. The test of dynamic changes is made by changing the structure of the routing to weaken the security control on SDS3 applications from SDS3-HalfSecurity to SDS3-NoSecurity.

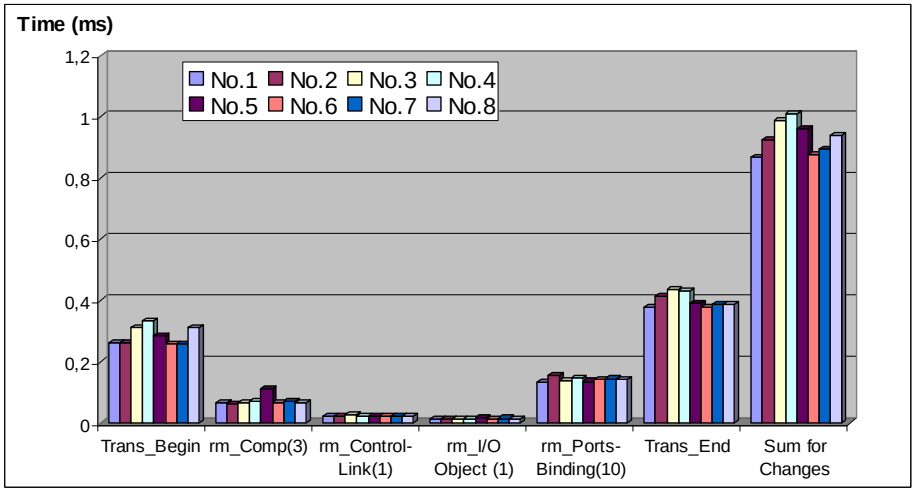

Figure 7.8: Running Time of Change from SDS3-HalfSecurity to SDS3-NoSecurity

There are series of operations involved in above dynamic changes: *transaction_begin()*, *remove_comonent()*, *remove_control_link()*, *remove_input_bject()*, *remove_ports_binding()*, and *transaction_end()*. The statistical test results of above changes are shown in Figure 7.8. The total time for the tested change is less than 1 ms, and most of time is consumed in the operation of *transaction_begin()* and *transaction_end()*. The changes time will vary in different change solutions that are comprised of different numbers and kinds of change operations.

7.2 Performance Analysis

Based on above performance tests of RBW, a detailed analysis on performance is given in the following two aspects which concern the execution efficiency and reconfiguration time of RBW and its potential applications.

7.2.1 Execution Efficiency Analysis

For the execution of RBW, most works are completed during the stage of initialization, such as routing schema parsing, routing instantiation and virtual binding. Only the real binding and unbinding operations are always followed with each request execution. All of these execution stages are closely related with the number of involved components and the number of communication ports of each component. Routing instantiation is seriously affected by the specific components because the sub-time of component instantiation is heavily decided by the parameters and properties handling of component. The virtual binding and real binding/unbinding are more concerned about the ports number of components. For ordinary requests to access applications, the execution managed by RBW is mainly comprised of three parts: real binding to load component, component execution for functionality, and real unbinding to unload component. To better explain the performance of RBW, we created a definition of component execution efficiency shown in Figure 7.9.

$$executionEfficiency = \frac{Time_{Execution}}{Time_{Execution} + Time_{Binding+Unbinding}}$$

Figure 7.9: Definition of Execution Efficiency

Components	Execution Time	Binding and Unbinding Time	Efficiency
Authenticator	23,315875(ms)	0,088454 (ms)	99,62%
Authorizer	7,2605(ms)	0,071517 (ms)	99,02%
Access Controller	1,24838(ms)	0,154873 (ms)	88,96%
Request Broker	0,1325(ms)	0,170099(ms)	43,79%

Table 7.1: Execution Efficiency of SDS3 Components

During the test of SDS3, the functional execution times and the execution efficiencies of SDS3 components are computed in Table 7.1. Although the times of real binding and unbinding are also varied with the ports number, the extent of this varying is still small, especially compared with execution times of components. Generally speaking, the execution efficiency is nearly decided by execution time of component. However, if the execution time of a component is also extremely small, the execution efficiency of the component would be rather lower. Based on above tests and analysis, the application utilization of RBW can be summarized as the following three categories:

- *Time Consuming Component Applications*: The RBW is recommended to be employed because the component execution efficiency will be very high, especially when the dynamic change is demanded in applications.
- *NOT Time Consuming and NOT Time Critical Component Applications*: The RBW could be employed in this case, but it is not highly recommended.
- *NOT Time Consuming But Time Critical Component Applications*: The RBW is not recommended to be employed in this kind of applications.

7.2.2 Reconfiguration Time Analysis

In the routing-level changes, there are two change operations which can be independently executed: cloning a routing and removing a routing. Cloning a routing will never disturb the current running application, and the running time to cloning a routing is affected by the specific routing. For example, under our testing environment the average running time to clone the routing *SDS3-HalfSecurity* is 9.5652 (ms). The operation of removing a routing will only affect the upcoming requests, not the ongoing requests. The running time to remove a routing is very small and independent of specific routings and specific components.

ID	Operation Name	Running Time	Comments [1]
op[1]	clone_routing()	9,5652(ms)	affected by Env., Routing
op[2]	remove_routing()	0,0959(ms)	affected by Env.
op[3]	transaction_begin()	0,25625(ms)	affected by Env., Routing
op[4]	insert_component()	0,0605(ms)	affected by Env., Comp.
op[5]	remove_component()	0,024(ms)	affected by Env.
op[6]	insert_control_link()	0,07838(ms)	affected by Env., Link
op[7]	remove_control_link()	0,02125(ms)	affected by Env.
op[8]	insert_input_object() insert_output_oject()	0,01213(ms)	affected by Env.
op[9]	remove_input_object() remove_output_oject()	0,0125(ms)	affected by Env.
op[10]	insert_ports_binding()	0,01403(ms)	affected by Env.
op[11]	remove_ports_binding()	0,01013(ms)	affected by Env.
op[12]	transaction_end()	0,53019(ms)	affected by Env., Routing

Table 7.2: Running Times for Change Operations Tested in SDS3

[1] affected by Env., Routing, Comp. Link means the runtime of the operation is respectively dependent on the running system environment, the specific routing, component, or control link. The current running system environment is the testing environment introduced in the beginning of this chapter. The current routing is *SDS3-HalfSecurity*. The current components are three secure components introduced in section 5.3, and the current control link is the AND Link used in SDS3.

The primary work of dynamic change in RBW concentrates on changing the structure of the running routing, which will result in a new collaboration relation of components without need to shut down the system. The change of routing structure is accomplished by two transactional operations and a series of change operations. In our test, the average running time of each single change operation is computed and shown in Table 7.2, in which running times are dependent and affected by different conditions clarified in the footnote. All the results are computed in the case of successful execution.

As introduced in section 4.4, the hard issue of consistency preserving is simplified to the synchronization of routing updating which is accomplished in the *transaction_end()*. In our tests, the average processing time for routing updating is only 0.00975(ms) which is the blackout time that users really have to wait for the dynamic changes. Different dynamic reconfigurations involving different change operations consume different processing times. Based on experimental results tested in advance, e.g. our test results shown in Table 7.2, we can analyze which change operations will be involved in a new reconfiguration and the processing time for the new reconfiguration could be predicated using the formula described in Figure 7.10, where $Time_{op[i]}$ indicates the running time of change operation op[i], and $Number_{op[i]}$ indicates the repeated times for this change operation.

$$Time_{predicted} = Time_{trans\_begin} + (\sum_{i=4}^{11} Time_{op[i]} \times Number_{op[i]}) + Time_{trans\_begin}$$

Figure 7.10: Definition of Predicted Time for Dynamic Changes

However, RBW is not originally designed for the components that hold persistent and varying states which are used and changed in component executions. Because in RBW each component may hold several component instances, different component instances are randomly selected for the execution of requests. The execution modeling of persistent components can also be addressed using a tradeoff solution in which only one component instance is instantiated for request executions and then the persistent sates can be changed with the same orders as the executions of requests. But the expense for this compromise is that the dynamic reconfiguration is not so efficient as we previously state. Because the transferring of component's persistent state has not been considered in our dynamic reconfiguration approach of RBW. To deal with the persistent component, the reconfiguration algorithm of RBW has to be modified and will generate the same issue as traditional approaches. In addition, the executions for persistent components are not so efficient because only one component instances will be instantiated and available for one routing or several routings to deal with the large number of requests.

Chapter 8

Related Works

The related works are introduced in this chapter. Firstly, the related works go to the approaches for dynamic reconfiguration which is the core issue addressed in this dissertation. Then, the related works are introduced for the issues that are involved in our two case studies: security control of middleware applications and web services dynamic composition.

8.1 Related Works on Dynamic Reconfiguration

As introduced in Chapter 3, the dynamism on distributed system is evolved from programmed reconfiguration, unplanned reconfiguration, adaptive system, to reflective system. The typical approaches concerning above different dynamism are summarized and analyzed in the following subsections.

8.1.1 Programmed Reconfiguration

ADL is a formal language to describe the static architecture structure of a component oriented software system. Dynamism is also supported on small number of ADLs or the extension of existing ADL to dynamically specify software architecture.

Darwin

Darwin [57] was proposed to specify static system structures in much common with other ADLs to describe a system as a configuration of connected component instances. However, through the use of conditional and replicated constructor, Darwin allows parameters to determine the system structure at initialization time [52], [53]. Further, Darwin has features to permit the description of dynamic structure which evolve as execution progresses. Structural evolution includes changes in both the connections between components and the set of component instances. Two mechanisms are designed to describe dynamic structures:

- *Lazy Instantiation*: The component providing a service is not instantiated until a user of that service attempts to access the service. Lazy instantiation is

modeled by using a dummy provision to which the clients of a server are initially bound. This dummy provision, in response to a binding request, returns the name of a prefix which triggers instantiation of the lazy instance and its associated bindings when communication is initiated.

- *Direct Dynamic Instantiation* permits the definition of structures which can evolve in an arbitrary way. Dynamic instantiation is modeled in the π-calculus by an agent that supplies the name of the instantiation service. This instantiation service triggers one of copies of a replicated process.

Dynamic Wright

Unlike the Darwin, dynamism of Wright [1] is achieved by extending a new notation to separate the dynamic reconfiguration behavior of architecture from its non-reconfiguration functionality. By providing a notation that provides a precise interpretation of each aspect, the description of architecture can be analyzed for the consistency and completeness. The approach idea to achieve dynamism in Dynamic Wright can be summarized as two parts:

- Special control events are introduced for description of component and port.
- These control events are used in a separate view of the architecture to describe how these events trigger reconfigurations.

In dynamic Wright the new notation for describing changes of architecture is called *Configuror*. By adding a *Configuror* to the notation and then clarifying what the control does, the designer gets a dynamic system. The *Configuror* consist of the following most important information:

1. When should the architecture be re-configured?
2. What should cause the architecture to re-configure itself?
3. How should the re-configuration be made?

8.1.2 Unplanned Reconfiguration

Unlike the programmed reconfiguration in which the reconfiguration is automatically triggered when the predefined conditions are met, unplanned reconfiguration is employed in the application where the reconfiguration operations are not predefined and is triggered by the administrator or program when necessary.

Kramer et al

In [51] Kramer et al proposed a change management model for system reconfiguration which had been widely used or extended by subsequent approaches for dynamic reconfiguration. In Kramer's approach the structural concern is firstly separated from

the concern of functional application, which makes the configuration of system explicit and enable to formulate general structural rules for change at the configuration level without consideration of application state. The proposed model for dynamic change management is shown in Figure 8.1 which enables to dynamically change the system from configuration *i* to configuration *i+1*. In addition, authors argued that the objectives of the dynamic changes should:

1. be specified in terms of the system structure;
2. be declarative;
3. be independent of the algorithms of application;
4. leave the system in a consistent state;
5. minimize the disruption to the application system.

Kramer et al firstly took the avoidance mechanism to ensure the reconfiguration leaving in a consistent state. During the change processing, the configuration manager identifies the set of components whose activities must be restricted to avoid leaving the system in mutually inconsistent states. Once these components have been identified, the manager instructs these components to be in passive state in which the component is only currently engaged in initiated transaction and will not initiate new transactions, so that safe sate for reconfiguration quiescence can be brought over the affected components.

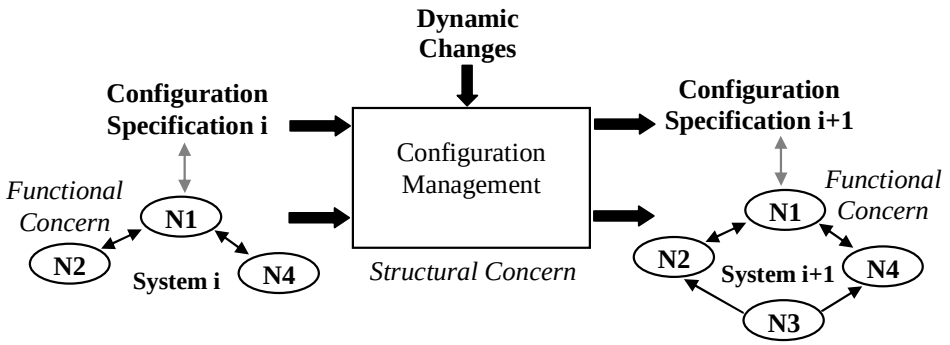

Figure 8.1: Change Management Model for System Reconfiguration

Goudarzi et al

The work of Goudarzi et al in [38], [66] is based on the Kramer's dynamic reconfiguration model and focus on maintaining component consistency during change processing. Kramer introduced a passive state to drive the component into the safe state. However, the defined rules for passive state are possible to result in deadlock in some case. Goudarzi et al proposed a new algorithm to allow all started transactions to terminate and ensure all components finally being driven into safe state. Compared to the passive state, unblocked state and blocked state are introduced in Goudarzi's

new algorithm. A coordinator sends a block message to all component nodes in the system. Once the component is in idle state, it will directly be moved into blocked state. But if the components moved into the blocked state receive a transaction request, they will be moved into the unblocked state. In this state, components continue their execution until they have serviced the request which unblocked them. At that time, they return to the blocked sate again. Unblocking a blocked component may well result in a new transaction before the unblocking sub-transaction request is serviced. Hence unblocking of a component will eventually lead to a fully blocked system. To enforce the safe sates of the component nodes that are directly affected by the changes, two measures are adopted (*BSet* for short to indicate the set of nodes which are targeted for blocking):

1. Dynamically expanding the *BSet* in step with outgoing transaction flow. In this way nodes received request from BSet members become members themselves.

2. Ignoring incoming request from non-set member.

Wermelinger et al

In [92] Wermelinger et al proposed a hierarchic architecture model for dynamic reconfiguration whose framework is also based on the model proposed by Kramer et al. Wermelinger aims to reduce the system disruption in two aspects:

- Minimizing the part of the system to be "frozen";
- Minimizing the time taken by reconfiguration operations;

Comparing the algorithms of consistency preserving from Kramer and Goudarzi, authors took a connection based algorithm to drive the affected components into the safe state in which the reconfiguration can be carried out. The essence of connection based algorithm is to block only those connections that will be removed. To block a connection its initiator node waits for any ongoing transaction to finish and then simply does not start a new one. Blocking connection means that the node will not serve any transaction that depends on the blocked one. To ensure that the blocking of one connection will not prevent other pending transactions to block, the configuration manager orders the blocking commands according to the dependency: if transaction t depends on *t1* then the block message is sent to the initiator of *t1* only after t is known to be blocking. This is always possible because transactions do not depend cyclically on each other. In addition, author separated the commands into different independent sets and define rules to enable execute commands in a parallel way which can reduce the disruption time considerably.

Rasche et al

Rasche et al presented a framework of Adapt.NET to enable runtime reconfiguration of component based application in [78], [79]. Adapt.NET is used for a web based remote laboratory - the Distributed Control Lab, to adapt experiment control for

failures in user control components. The adopted dynamic reconfiguration model and algorithm to preserve consistency are based on the work of Wermelinger et al [92] which is extended from Kramer et al [51]. The features and contributions of Adapt.NET can be summarized as follows:

1. The component configuration is identified as an aspect of the system. Aspect Oriented Programming (AOP) technology is employed to implement the interface *IConfigure* to realize the configuration functions, such as transaction handling, connections and starting/finalizing the component etc. Using AOP the configuration functions can be weaved into the binary component, e.g. .NET components etc.

2. The communication between application components is realized via connectors. Through introducing new connector types, such as local call connector and IIOP connector etc., the Adapt.NET is able to adapt the components deployed on heterogeneous platform, such as Microsoft .NET, CORBA and J2EE.

3. Authors analyzed in detail the blackout time and reconfiguration time during the dynamic reconfiguration on components, and gave an experimental evaluation on the two kinds of time consuming in different situations.

Almeida et al

In [2] Almeida et al proposed a dynamic reconfiguration service for CORBA that allows the reconfiguration of a running system with maximum transparency for both client and server side developer. The adopted dynamic reconfiguration model is also from the Kramer [51] with series of extensions for consistency preservation etc. Almeida applied the dynamic reconfiguration model to CORBA object and gave a series of concrete sub-solutions for typical issues as follows:

- *Structural Integrity*: In CORBA the structural integrity is embodied as referential integrity and interface compatibility. Referential integrity becomes an issue whenever an object reference changes. In order to re-establish the reference binding after reconfiguration, authors provided a location agent for clients to find the objects with invalidated object references. To realize interface compatibility, the new object must implement the old interface or an interface derived from it.

- *Mutual Consistency*: To guarantee the safe state of affected CORBA objects, authors distinguished the requests into three sets: i) blocking set for requests that prevent affected objects to reach safe state; ii) laissez-passer set for request necessary to reach safe state; iii) requests that do no involve any affected object. In implementation a selector was designed to determine whether the request belongs to the laissez-passer set or not. If yes, it is forwarded to target object as in normal operation. Otherwise, the request is sent to the blocking queue and will be redirected to the new version of target object after the reconfiguration.

Cao et al

In [15] Cao et al proposed a graph-oriented model for constructing reconfigurable distributed programs. Based on proposed model authors implemented an integrated software platform, called Distributed Implementation Graphs (DIG), which provides a high-level logical graph construct and a collection of software facilities to support graph-oriented distributed programming. In a graph-oriented distributed model, a distributed program is defined as a collection of local programs (LPs) that can execute on multiple processors, and the LP represents the parallel computation. The communication between LPs is via message passing. To achieve the dynamic reconfiguration on components mapped to graph-oriented model, a series of primitive operations are provided: such as *AddVertex*, *DelVertex*, *AddEdge* and *DelEdge* etc. A dynamic reconfiguration manager is also presented to be responsible to validate a reconfiguration plan and to coordinate the distributed processes to execute the plan. A central server based prototype is implemented on top of a Parallel Virtual Machine (PMV) which handles massage routing, data conversion and task scheduling across a network of heterogeneous platform.

The most important contribution of the approach is that it provides powerful capability to model the dynamic features of various kinds of distributed and parallel programs. However, authors did not address well the issue of consistency preserving during reconfiguration processes which is in particularly critical in the of graph oriented modeling for distributed components.

Shivastava et al

In [83] technology of transactional workflow was employed to realize application composition and execution environment in which the dynamic reconfiguration on application structure is supported. In author's workflow based approach the application components are modeled by a task model in which a task is an application specific unit and a task controller is designed to supervise and control the activities of the task. More specifically,

- *Task* i) has alternative input sources to acquire a given input from one or more sources; ii) has alternative outputs to produce specified output object; iii) can be composed from other tasks to form a compound task; iv) has a detachable implementation to contain application codes.

- *Task Controller* is exclusively created for each task to record the persistent dependencies and manage the execution activities. The task controller receives notifications from other task controllers and uses this information to determine when its associated task can be started.

The execution of a workflow application is controlled by the exchange of notifications between task controllers and tasks. Through task modeling the functional component is separated from its dependencies and execution environment, which can enable flexibly and dynamically change on the components. The use of transactional

workflow ensures the changes can be atomically carried out with respect running application. The consistency issue is dealt with by checking the states of all involved task controllers at the beginning of the transactional change operations, and there is no new algorithm taken for consistency preserving because it is not the key point of this approach.

8.1.3 Agent Based Reconfiguration

Agent technology is used in lots of approaches to enable dynamic reconfiguration on distributed component system, in which the agent is used to construct the application component or the system deployed in remote host to monitor the components.

Palma and Bellissard et al

In [7], [74] authors presented a system that enable dynamic reconfiguration on the agent based applications. In author's agent based distributed system, the reconfigurable application components is constructed as a agent which follows an execution model based on event. An event received by an agent triggers a reaction which itself may send out new events. Signaling event is the only mean of communication between agents. Events are delivered to the agents via a message bus called Channel. In the agent based system there is a execution engine which provides a loop program to successively get the notifications from the message queue and calls the relevant reaction function member of the targeted agent. To perform the reconfiguration actions during running time, a Configurator Agent is created to take charge of transmitting reconfiguration actions to either the application agents for a new assignment, such as interconnection update, or to the message bus, such as agent removal and creation etc. The reconfiguration actions on the agents are dealt with the simple mechanism as follows:

1. Freeze the agent and rollback its state to save on disk;
2. Remotely create a agent of the same class and initialize its state according to the saved one;
3. Change every role that was bound to the initial agent and replay the possible suspended reaction with new agents.

To preserve the consistency, authors define constraints for each reconfiguration, and define three states for agent execution, similarly in [38]: active state, passive state and frozen state.

Cherif et al

Cherif et al also developed a distributed software architecture based on agent which has the ability to react to events and perform architectural changes autonomously [77]. The distributed intelligent components in the architecture are called Agents which act

autonomously to adapt dynamically the application without outside intervention. In a distributed application across several locations multiple agents are provided. Each agent monitors one local application and communications with other agents over network. To order to react with the architecture and the architectural environment, the events, conditions and action rules have to be assigned to each agent. The dynamic operations of the agent based architecture are classified into Agent_Primitive which is made up of the primitive operations, and Agent_Strategy which are the composite operations that call upon the primitive operations. The operations for Agent_Primitive contain: creating component, connector, role and port, adding port to components, adding role to connectors, and even setting/getting a value of attribute quality of components, connectors etc. The operations for Agent_Strategy contains: adding component/connector to architecture, deleting component/connector from architecture, migrating component to another agents, etc. Author's agent based architecture presents a powerful and flexible dynamic change management for the components distributed across different locations. However the approach to address the issues involving in change process did not explicitly explained.

Castaldi et al

In [18] Castaldi et al presented a system of Lira which is a lightweight infrastructure for managing dynamic reconfiguration on component based distributed system. The approach of Lira is to define a series of methods to apply the basic facilities of Internet-Standard Network Management (ISNM) to the complex component-based software system. The primary modules developed in Lira are summarized as follows:

- *Reconfiguration Agent* is associated with a component and is responsible to reconfigure the components in response to operations on defined variables. The reconfiguration agent takes charge of the life cycle of its component by set of functions, such as *start()*, *stop()*, *suspend()*, *resume()* and *shutdown()* etc.
- *Host Agent* is associated with a computer in the network and is responsible to install and activate components on that computer in response to requests from a manager which can also be a reconfiguration agent. In the process the host agent provides an available network port, called agent address, to the reconfiguration agent over which that agent can receive request from a manager.
- *Management Protocol* follows the SNMP paradigm. Each message in the protocol is either a request or response which is transferred between agents and managers.

8.1.4 Adaptive Systems

In adaptive system the changes are automatically carried out to adapt the application system to varying environment or requirements. So more functions are needed to complete: predefining the change conditions, monitoring and evaluating the execution, and carrying out the change operations.

Oreizy et al

In [73] Oreizy described a proposal of architecture based approach to support comprehensive adaptation on software system from adaptation-in-the-small to adaptation-in-the-large. There are two facilities to consist the architecture based approach: adaptation management and evolution management. Adaptation management is responsible to observe and analyze its behaviors and the changing circumstances. To be more specific, the adaptation management consists of the functions as follows:

- *Collecting Observation*: large numbers and varieties of observations and measurements are required, such as embedded assertion within the application, expectation agent for modeling application behaviors etc.
- *Evaluating and Monitoring* representative behaviors of the running system are required to be monitored to maintain the consistency whose management requires hybrid approach to combine both static and dynamic analysis.
- *Planning the Changes* have two forms: observation planning to determine which observations are necessary for deciding when and where adaptation are required, and adaptation planning to determine exactly which adaptations are made.
- *Deploying Change Descriptions*: changes agents is required to propagate change description among sites to coordinate the change dispatched from third site.

Evolution management is responsible to evolve the application system in a consistent manner, and consists of the functions as follows:

- *Dynamic software architecture*: two approaches are adopted and blended into one single cohesive whole: C2 which is optimized for flexible components, and Weave which focuses on high-performance and flexible connectors.
- *Maintaining Consistency and System Integrity*: an Architecture Evolution Manager (AEM) is proposed to mediate all change operations directed toward the architecture.
- *Enacting Changes*: a visual, interactive and architecture editor can be used to construct architecture and specify modification.

David et al

David et al presented an infrastructure which enables to make middleware platform adaptable to changing execution environment [23]. The adaptation is achieved by dynamic reconfiguration on the associations between functional components and non-functional services. The infrastructure consists of the following main parts:

- *Application Model*: the application is composed of two kinds of components: functional components fulfilling the application functionality, and the non-functional component to realize the management services, such as distribution and persistence etc.

- *Observation*: two kinds of functionalities are realized in the framework of observation: i) exposing enough information to adaptation engine to make the decision; ii) detecting meaningful change in above information which will result in adaptation.

- *Adaptation Engine* is the heart of the system which makes the adaptation when meaningful change is detected. The adaptation engine is specialized for each application by configuring adaptation policies. The adaptation policies can be split into low-level system policies whose role is to define adaptation rules independently of application semantic, and higher-level application policies which are designed for application programmer.

Mukhija et al

A Contract-based Adaptive Software Architecture (CASA) was developed by Mukhija et al in [67], to enable dynamically adapting the functionality and performance in response to runtime changes in their execution environments. The constituent entities of the CASA framework are described as follows:

- *Applications* are composed of a series of components to provide the functionality. Apart of the component, another two entities are the Application Contract and Service Negotiator. The application contract of an application in divided into operating zones each of which contains a list of valid alternative component configurations and corresponding resource requirements. The service negotiator is responsible to negotiate the offered QoS with the expectation of its peer applications.

- *Contract-based Adaptation System* (CAS) is the entity to carry out dynamic adaptation on behalf of its associated application.

- *Contract Enforcement System* (CES) is responsible for satisfying resource requirements of all applications running on its host node.

- *Resource Manager* (RM) monitors the value and availability of resources and keeps the CES updated with the current resource status.

In [68] authors also gave an example to explain how the CASA is applied in a self-organized mobile network environment to enable the interaction processes and dynamically adaptation on the mobile nodes.

8.1.5 Reflective Systems

The reflective system provides more flexibility than dynamic reconfigurable system because it enables to exhibit meta-data information of the system to third-party application, which offers the possibility for third-part program to make dynamically reconfiguration on the system.

OpenORB

Blair et al applied the concept of reflection which originated from the area of programming language, to develop a reflective middleware - OpenORB [10]. Blair et al argued that a reflective system should have two features:

- *Inspection* - reflection can be used to inspect the internal behavior of a system, so the third part program can also employ such functionalities to implement the functions, such as performance monitor and QoS monitor etc.

- *Adaptation* - reflection can be used to adapt the internal behavior of a system, which can satisfy the requirement of system running in varying environment.

In author's approach of reflective OpenORB the core technology is the open binding illustrated in Figure 8.2. The open binding is composed of a series of component objects, binding objects and local bindings which connect the component objects and binding objects. The binding object could also be an open binding which then results in a recursive binding structure, or a primary binding whose implementation is closed. For example, the primary binding could be a Real-time transport protocol to bind two distributed object. The component object may the MPEG encoder/decoder or the QoS monitor/controller etc. Through the control interface user is able to inspect the structural information of open binding and make adaptation on the structure. Authors' reflective middleware - OpenORB is based on a CORBA implementation from Sun Microsystems, called Cool-ORB. The implementation of open binding is based on a communication system - Ensemble which enables the programmer to select a particular protocol profile at binding time and supports the running time modification and reconfiguration of modules.

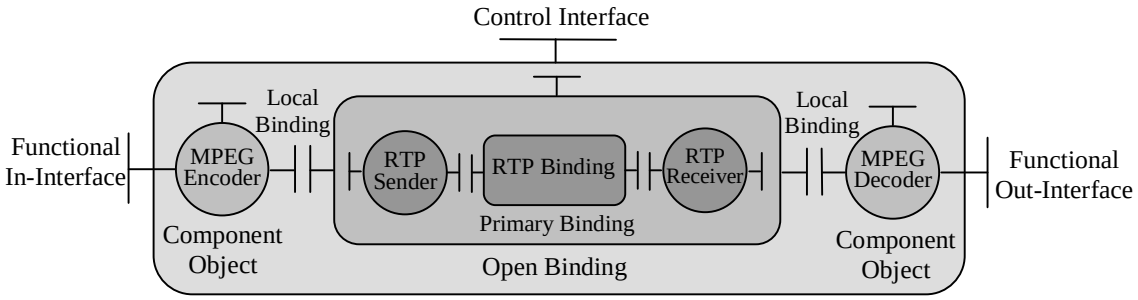

Figure 8.2: A Nested Open Binding

DynamicTAO

Kon et al developed a dynamically adaptive middleware, called DynamicTAO [50] which is based on the existing configurable middleware TAO. DynamicTAO is said a reflective ORB because it allows inspection and reconfiguration of its internal engine. The reflection in DynamicTAO is achieved through a collection of entities known as

the *Component Configurator* which holds the dependencies between a certain component and other system components. There are three kinds of entities that employed *Component Configurator* for dynamism management. Each process running dynamicTAO contains a customized *Component Configurator*, called DomainConfigurator which is responsible for maintaining reference to instances of the ORB. A TAOConfigurator maintains the reference of dynamicTAO strategy implementations, and ServantConfigurator holds the references of servant instances attached in the running ORB.

Authors developed multiple configuration tools, such as network broker and reconfiguration agents. All the reconfiguration requests will be forwarded to a dynamic (service) *Configurator* that holds all kinds of *Component Configurators*. The operations of dynamic *Configurator* can be classified into two kinds. One kind of operations in the interface are used to inspect the dynamic structure of system, such as *list_domain_components()*, *list_loaded_implementations()* etc. The rest operations are used to dynamically adapt the system, such as *load_implementation()*, *delete_implementation()* etc. All the loadable component implementations are organized in categories representing different aspects of the TAO ORB packaged as dynamically loadable libraries that can be linked to the ORB at run time.

mChaRM

A reflective middleware mChaRM, which extend the Java RMI, was developed by Cazzola et al in [19]. In mChaRM a new reflective model, called the multi-channel reification, was presented and complemented to address a series of problems arose by distributed application developing, such as security, communication reliability and multi-communications etc. The multi-channel reification model is based on the idea of considering a message sent through a logical channel established among a set of objects requiring a service, and a set of objects providing such as service. The logical channel was reified into a logical object, called multi-channel, which monitor message and enriches the underlying communication semantics with new features used for the specific performed communication. Author's reflective middleware mChaRM mainly consists of three parts:

1. A preprocessor which transforms the client invocation described by self-defined language into the Java syntax.

2. A Java package which contains the basic classes needed to develop all kinds of multi-channels demanded for different requirements.

3. A Java package which contains some implemented examples of multi-channels.

The reflection is embodied in the functions of introspection and intercession which are carried out on the messages dispatched through a multi-channel, such as getting/setting method name, inspecting, modifying or removing arguments etc.

8.1.6 Comparison and Evaluation

The approaches for programmed reconfiguration, such as Darwin [57] and dynamic Wright [1], do provide the capability to specify the component oriented software architecture dynamically. However, all the changes on the software structure have to be pre-designed and compiled before system is released, that only meet the need of small partial applications in which dynamic changes are demanded.

In most applications, people don't know which component will be changed and what change operations has to be made. So a series of approaches were proposed to enable the unplanned changes on the system structure. The pioneering work was done by Kramer et al [51], who separated the structural concerns from the functional concerns and proposed a change management model for system reconfiguration. Kramer's change management model was frequently referred, improved and extended. For example, Goudarzi [38] and Welmelinger [92] improved the algorithm of consistency preserving during the dynamic changes, Almeida [2] applied the improved model to CORBA and proposed a practical solution for changes on CORBA applications, and Rasche [79] employed the model to develop a framework which allows to carry out dynamical changes on the applications deployed in heterogenous platform. All above mentioned approaches inherits a common shortcoming: adopting a strategy of *waiting until safe state*, refer section 3.5 for details. If the algorithm of consistency preserving is improved, just as the work of Goudarzi and Welmelinger, the reconfiguration time does decrease in certain extent. However, this kind of change model has to face and can not address this issue: if the component affected by change keeps in a long time interaction, the change processing has to wait long time. In RBW, this issue can be avoided because the changes is acted on an idle execution environment, namely routing. The reconfiguration time can be predicted and is only lightly affected by the structure of specific components. More importantly, the block time that subsequent requests really have to wait is simplified as the time for routing updating which is always limited in a extremely small value, refer section 7.2 for details.

In Agent based approaches for dynamic reconfiguration, agent may be composed as the functional component [7], [74] or be composed as system to monitor the component [77], or even act as the non-functional management [77], such as reconfiguration activities. The main contribution of agent technology for change management is that it allows to coordinate and dynamic change the components deployed in distributed network. In RBW, I also proposed an extension for distributed components, in which components deployed in different computers can be managed or even dynamically changed by RBW, refer section 4.6 for details.

Adaptive and reflective approaches enrich the functionality of dynamic reconfiguration to enable they can be used for special applications. Adaptive systems [73], [23] enlarge the change management with monitoring and evaluating functionality which enable to make the application system automatically adapted for the varied environment. The reflective systems [10], [50] enlarge the change management with inspection of meta-structure of application system which enable third-party program to make dynamic changes on the system structure according to the special demands. In addition, both agent based technology and adaptive, reflective approaches didn't provide

solution to address the issues arising from dynamical changes.

There are some approaches which are proposed to model the execution and management of collaborative components and also result in the capability of dynamic changes. For example, Cao et al [15] proposed a graph-oriented model to construct distributed components, in which the components are mapped into the node of graph-oriented model and the dynamic changes are realized by changing the structure of graph-oriented model. However, above mentioned issues were addressed. Shivastava et al [83] employed a transactional workflow technology to model the execution environment of functional component, in which a task controller is designed to supervise and control the activities of component. Our RBW is much affected by this transactional workflow modeling. However, in Shivastava's approach the task controller is tightly fixed with functional component, which lead to less flexibility than RBW. In addition, the execution and management modeling of RBW realize the concept of temporary binding which results in a high dynamism that can not be reached in Shivastava's approach.

8.2 Related Works on Middleware Security

For the popular applied middleware system or products, such as CORBA, J2EE and DCOM/.NET etc., already provide powerful security mechanism implementations and rich set of security APIs to help developer create secure applications over Internet. So most research works concerning middleware security are involved to extend the existing solution or create their own platform to provide additional or special application demanded security functionality. Most extensions of existing solution are based on CORBA for its popular acceptance and lots of open source implementations. Some extended security functionalities are for special application demands, such as multi-level security interoperability [48]. Some approaches provide efficient security management, such as mechanism to easily deploy application-specific security policy [8]. With the emerging of new Internet applications, there are also research works trying to address the security in e-commerce transactions [70]. In addition, the security interoperability between the system supported by different middleware technologies is also investigated [31].

Kang et al

A practical and CORBA based secure solution was proposed by Kang et al in [48] to achieve Multi-Level Secure (MLS) interoperability in which the access control decisions are made based upon the level of user's trustworthiness determined by enterprise administrator and sensitivity of the accessed information. Authors' MLS architecture is constructed from multiple conventional single-level CORBA and two special security devices which are described more as follows:

- *Starlight Interactive Link* which uses high-side proxy and low-side proxy to mimic the real low-side client and high-side server to realize the data commu-

nication from high side to the low side, and eventually enables high level users to access low level resources.

- *NRL Pump* is a one-way device provides a secure way to replicate low-level information for higher level applications and users.

In the CORBA-based approach implementation the NRL Pump is configured as a CORBA object and multi-level Starlight-client is configured as CORBA clients.

Beznosov et al

Beznosov investigated and developed a framework to reason about the middleware security mechanism which enables application-specific access control [8]. Firstly author classified the middleware providing secure functionalities into three catalogues:

1. MDME - every security functions is done by middleware that has no application-specific security;
2. ADME - Only application decisions are enforced in middleware;
3. ADAE - Both decision function and enforcement function are implemented in application, not in middleware platform.

Based on the ADME schema, authors proposed a new approach to realize application-specific access control in middleware, in which two key techniques are as follows:

- *Object Security Attributes* (OSA) was introduced to generically represent application specific factors in middleware application. The semantic interpretation of an OSA is completely up to the processing entity. Example could be name-value pairs or XML-based structure etc.
- *Attribute Function* (AF) which is independent from the concrete application, is introduced to communicate between Enforcement Function (EF) in middleware and Decision Function (DF) in application objects. Moreover, the implementation of AF is provided by the application, not by middleware platform.

An implementation of such approach based on OSA and AF was realized on CORBA as an extension and improvement to the CORBA Security.

Syntegra Federal

Unlike other current secure middleware to realize security in middleware system level or application level, Syntegra Federal proposed a software solution of Secure Access Middleware (SAM) [29] to protect the data in the level of data repository level. The key technology to enable repository level protection is realized by inserting a new layer of control between open interfaces and product-specific proprietary code. SAM offers protection across COTS-based data stores and provides a foundation of granularity supporting intruder detection and dynamic alerts by a series of inserted security service modules. From the viewpoint of security application, SAM offers the following features:

- Offer a single point to protect the access on the data stores independent of applications.
- Provide user-specific access operation to the object and ensure appropriate authentication credentials.
- Selectively encrypt content using keys produced by the SAM to ensure interoperability across different applications.
- Support popular ODBC database and LDAP directory products whose data driver can be modified to intercept transactions and perform security checks prior to submitting the transaction to the data server.

In the implementation, most of security functions are realized by different Service Modules (SM), such as Authentication SM, Access Control SM and Auditing SM etc.

Nenadic et al

A secure middleware project - Fair Integrated Data Exchange Services (FIDES) was developed by Nenadic et al [70] to support e-commerce transactions and provide series of security services demanded by e-commerce. FIDES was designed to satisfy the security requirements:

1. strong fairness for both parties joining the exchange process;
2. Non-repudiation for each participant during the process;
3. confidentiality of the exchanged items;
4. Reduced role and transparency of the Semi-Trust Third Party (STTP).

Two FIDES protocols are designed to enable secure e-commerce transaction: a normal exchange protocol performed by two business parties and a recovery protocol performed between STTP and business party. The symmetric encryption algorithm and public-key algorithm are employed in two protocols for the functions of secure exchange and non-repudiation. The FIDES system consists of three entities:

- *FIDES Server* is the core of the system consisting of transaction manager and protocol library, JMS and secure storage etc.
- *FIDES Client* provides a GUI-based application interface that allows a business user to securely access the FIDES services.
- *FIDES STTP* provides an online facility for dispute resolution and recovery of exchanged item, in cases when a normal exchange process fails.

Foley et al

A secure WebCom which is a secure distributed computing architecture was developed by Foley et al [30], [31] to enable security interoperability between the COM+/.NET, CORBA and Enterprise Java Bean. In secure WebCom the RBAC is employed for the privilege management on the application objects. A technology of KeyNote which is an expressive and flexible trust management schema was used to help coordinating the trust relationships between the different system and their associated security policies. The secure WebCom environment can automatically convert middleware RBAC policies to their equivalent KeyNote policies, and vice-versa. This enables a high degree of policy interoperability between different middleware systems. The policy based trust coordination can be divided into follow sub-tasks:

- *Policy Configuration* whose task is to translate the specified KeyNote RBAC policy into the equivalent middleware RBAC security configuration.
- *Policy Comprehension* whose task is to transform the middleware security configuration back to KeyNote RBAC policy, which needs the understanding of the entire policy in one common format.
- *Policy Migration* is needed for the interoperability of security policy when existing or modified policies have to be moved from one middleware system to another system.
- *Policy Maintenance* is necessary to preserve a consistent global policy across the different heterogeneous middleware systems.

8.3 Related Works on Services Dynamic Composition

The dynamism on web services composition concerns two aspects. Firstly web services are dynamically selected and automatically composed into the composite service. Although workflow based composition approaches may also provide such dynamic capability, such as eFlow [16], semantic based approaches are more suitable to offer dynamic selection [35]. Secondly dynamic change on the structure of composite service. In most approaches the changes are made on the schema of composite service [17], [95] which will be executed at runtime to achieve dynamic composition. The approach based on dynamic business rules [21] is distinctive from others for its usage of interpreted AOP language. The second kind of dynamism is also reached in our DSC, in which the changes are not directly made on the schema, but on a instantiated schema. Moreover, the typical issues arose by dynamic change, which is not explicitly mentioned in the following approaches, has been successfully simplified for the employment of RBW. In fact, the RBW could be integrated with other approaches, such as semantic technology, to achieve more powerful solution for dynamic and automatical service composition.

Casati et al

An eFlow was developed in HP [16], CIJKS002 to support the specification, enactment and management of composite e-services, which can be preassembled or created on the fly, and can dynamically adapt to changes in the business environment. In eFlow the composite service is modeled as graph-oriented service process, including service, decision and event nodes, which can be instantiated several times and be concurrently running. The dynamism of eFlow is behaved in the following aspects:

- *Dynamic Service Discovery*: a service selection rule, which has several input parameters, is designed in service node to enable dynamically discover and select the appropriate service upon the customer's requirements.
- *Multi-Service Nodes*: a special service node, called multi-service node, is designed to enable invoking multiple, parallel instances of the same type of service ,and the number of service nodes to be activated is terminated during run time.
- *Dynamic Service Node Creation* is supported in eFlow by a generic service node which is not statically bound or limited a specific set of services. The configuration parameters with a list of service nodes can be set at instantiation time or at run time.
- *Ad-hoc Changes* are modification applied to a single, running process instance. The ad-hoc changes are first made on the process schema by authorized users and then migrated to the running process instance. Some consistency rules are specified in eFlow to guarantee the behavioral consistency while migration.
- *Bulk Changes* is to apply changes to many instances of the same process. With this mechanism the set of instances with common properties can be changed in one operation, instead of case by case.

Zeng et al

In [95] Zeng et al developed a DYnamic intelligent flow for web services composition, called DY_{flow}, in which end users can define their business objectives and the system dynamically composes web services to execute business processes. In DY_{flow}, web services, business objectives, business rules are specified over defined set of services ontologies which adopt a common language to define basic concepts and terminologies in a community. The dynamic features of DY_{flow} are embodied in the following aspects:

- *Business Rule Templates* are proposed to facilitate the description of business policies. There are two kinds of business rule templates. One is service composition rules which are used to dynamically compose services. Another is service selection rules which identify a particular algorithm or strategy to be used for choosing a web service to execution task during the runtime.

- *Adaptive Service Composition Engine* which enables runtime modification of composite service schemas. Users can define business rules to modify the composite service schema in DY_{flow} at any time. The composition engine can automatically incorporate newly added business rules at runtime if necessary.

Fujii et al

A semantic information based approach for dynamic service composition was proposed and implemented in [35]. To enable semantic-based dynamic service composition, both the modeling of service components and the service composition mechanism are required to support semantics. Author proposed semantic approach is comprised of three parts: a Component Service Model with Semantic (CoSMoS) to integrate the semantic information of a component and the functional information of component into a single semantic graph representation, Component Runtime Environment (CoRE) to convert different component implementation onto CoSMoS representation, and Graph based Service Composition (SeGSeC) to perform semantic matching. The details of three parts are described as follows:

- *CoSMoS* is an abstract model that represents a component as a single semantic graph. CoSMoS defines component in four domains: Data Type Domain for data type of component, Semantic Domain to represent abstract ideas or actions, Logic Domain for logics knowledge about component, Component Domain as a combination of its data type, logics, operations and properties.

- *CoRE* provides discover interfaces and access interfaces for components implemented in various component technologies. The discovery interface provides the functionality to discover a component its keyword, URI, property or operation. The access interface provides the functionality to invoke an operation of a component.

- *SeGSeC* allows a user to request a service using a natural language sentence, and generates the execution path, which describes the structure of a composite service, of the requested service, and executes the composite service.

In author's semantic approach the composite services are always dynamically composed at the running time according to the semantic information. The changes of the composite service are realized by changing the semantic information of component.

Wu and Vukovic et al

An AI planning based approach, Simple Hierarchical Ordered Planner 2(SHOP2) [69], was employed for automatic and adaptive web service composition. The SHOP2 is a domain independent planner, which uses a Hierarchical Task Network (HTN) to decompose an abstract task into a group of operators that forms a plan to carry out the task. In SHOP2 the planning system decomposes tasks into smaller and smaller subtasks until the primitive tasks are found that they can be performed directly.

Wu et al [94] believe that the concept of task decomposition in HTN planning is very similar to the concept of process decomposition in DAML-S process ontology. Authors gave a very detailed description on the process of translating DAML-S to SHOP2. In their implementation, the following modules are included:

- A DAML-S to SHOP2 translator which translates a collection of DAML-S process definition into a SHOP2 domain.
- An interface to let users specify the request for a service.
- A monitor which handles SHOP2's calls to external information collecting web service during planning.
- A SHOP2 to DAML-S plan converter which convert a plan to DAML-S format.

In [91] Vukovic et al also presented an architecture which enable specific context aware dynamic service composition using the SHOP2 planning system and BPEL4WS technology. In authors' implementation the SHOP2 planner is employed for selection of required services and the sequence of services execution, and the BPEL4WS is used to express the logic of a composite service.

Cibran et al

An approach of business rules was proposed by Cibran et al [21] for web service dynamic composition. A business rule is defined as a statement to define or constrain some aspect of a business service, which is intended to assert business structure or control the behavior of the business. Authors' approach is based on existing composition specification - BPEL4WS, and the business rules are defined and imported to trigger the event or conditions of the execution of the core application. These rules will decide whether to add, replace, change or remove activities that are present in the core composition.

Different business rules are categorized and illustrated in detail with examples and solutions. The implementation of business rules are realized using an AOP language, JAsCo, with which the business rules are treated as aspects to be inserted and integrated with the core composition. In authors' interpretation based approach, the aspects can be plugged in at runtime when the interpreter reaches their pointcut definition. So the dynamic composition is achieved when business rules are plugged in/out as aspects at run time.

Chapter 9

Conclusion and Future Work

In this last chapter I comes to a conclusion of my work by summarizing its advantages and disadvantages, and outlines the future work.

9.1 Summary of the Advantages

Dynamic reconfiguration is strongly demanded in long time running systems where a break of service is not tolerable, as well as in mobile systems where the service has to be adapted dynamically to meet the varying client environments.

In this dissertation I proposed a technology named Routing Based Workflow (RBW), which is designed to model the execution and management of collaborative components. In RBW, an XML based configuration language was designed to specify the properties and IO behaviors of each component and the interactions among different components. More importantly, through modeling the execution environment of collaborative components, a high flexible execution management of components is achieved. In RBW, the execution management of components is separated into two phases: virtual binding to create a connected idle execution environment for collaborative components, and real binding to guide the executions of series of components. Through separation of real binding from virtual binding, temporary binding for component instances is realized. The temporary binding means the component instances are temporarily loaded into a created execution environment to execute their functions, and then, released to their repository after execution. The advantages of RBW can be summarized as follows:

- RBW allows to model the execution of collaborative components from their static structure to running time state.
- RBW allows to model different kinds of flow structure of collaborative components, such as sequential and parallel structure, flow pick structure, and cycled structure.
- Control management on the collaborative components, such as suspend, resume, stop operations etc., can be easily carried out in RBW with the control depen-

dencies that are automatically created according to the data flow of components supplied by RBW schema.

- RBW addresses the problem of dynamic reconfiguration by its distinctive execution management - temporary binding for component instances, through which the hard issues, such as consistency preserving etc., are greatly simplified. The reconfiguration time can be predicted with knowledge of experimental data tested in advanced, and the blackout time can always be kept in an extreme small value in any applications.

- Multi-solutions are supported synchronously in RBW, through which RBW can be employed to develop the system providing personalized services.

- An extension of RBW is able to model and manage the execution of collaborative components which are deployed separately.

With the technology of RBW, a secure middleware system - Smart Data Server Version 3.0 (SDS3) is developed on top of CORBA. In SDS3, partial CORBA is adopted and modified as the underlying communication infrastructure, and three secure components, e.g. Authenticator, Authorizer and Access Controller, are developed and managed by RBW to enhance the security control on the invocation of deployed applications. The SDS3 not only demonstrates the feasibility of RBW, but also provides a framework to easily develop the applications that need be protected. The advantages of SDS3 can be summarized as follows:

- Multi-levels security control is provided in SDS3 from coarse-level security strategy control to detail-level application specific security control.

- The coarse-level security strategy control can be dynamically changed by reorganizing the secure components, which is a feature inherited from RBW.

- The detail-level application specific security control is achieved by configuring the policy for component of Access Controller. So the application of SDS3 contains only business processing logics and does not need to consider any issues of security in source code.

- Multiple security control strategies are supported synchronously in SDS3 to enable flexible security control for different kinds of applications, which is a feature inherited from RBW.

As another case study of RBW, Dynamic Service Composer (DSC) is developed on top of Apache Axis and Apache WSIF. In DSC, RBW is employed to integrate several internal or external web services into a composite service, and Apache Axis acts as the underlying platform to deploy and publish the self-developed single web services and the composite services. Apache WSIF is used to help constructing the Service Invoker which carries out the function of web service invocation and is managed by RBW. Firstly, DSC demonstrates that RBW can be used for web services composition, which

is a promising approach for enterprise applications integration. Secondly, compared to other approaches for web services composition, DSC has the following advantage inherited from RBW:

- DSC is able to dynamically change the service providers and change the composition structure to provide better service.

9.2 Summary of the Disadvantages

The temporary binding for component instances brings much flexibility for management of collaborative component. However, it also leads to the issue for execution efficiency because the extra operations of real binding and unbinding appear in each request execution. Fortunately, according to our analysis the operations of real binding and unbinding are composed of a series of simple manipulations, such as setting/removing the object references etc. The performance tests also show that the temporary binding pose very small impact on the execution of components that consume the time for functional execution much higher than the time for real binding and unbinding. However, there are also some kinds of components which consume a little time for its functional execution. If this kind of component will be used for time critical application, RBW is not recommended for such application because the execution efficiency is low in this case.

In addition, RBW is not originally designed for the components that hold persistent and varying states which are changed in component execution and will be used in subsequent components. Because in RBW each component may hold several component instances, different component instances are randomly selected for the execution of requests. For the components with persistent state, the modeling can also be addressed with a tradeoff solution in which there is only one component instance instantiated for request executions and so the persistent sates can also be changed with the same order as the executions of requests. But the price for this tradeoff solution is that the dynamic reconfiguration is not so efficient as we previously state. Because the transferring of component's persistent state has not been considered in our dynamic reconfiguration approach of RBW. To deal with the persistent component, the reconfiguration algorithm of RBW has to be modified and will generate the same issue as traditional approaches.

For the two application cases: SDS3 and DSC, it is hard to list their disadvantages. As experimental systems, they surely have numerous works, e.g. usability and stability etc., that need to be done to become products. In particular, DSC is just a demo system for the application of RBW in web services composition.

9.3 Future Work

As introduced in section 4.6, the RBW extension can also be used to model and manage the collaborative components distributed separately. However, an investigate has to be carried out in a real application environment, e.g. mobile computing, to

ensure which new issues will arise, and how to address such issues. Moreover, I hope to apply the RBW extension in a practical application system in the future.

In addition, I hope to combine our RBW with the widely accepted web service composition specification, such as BPEL4WS, and semantic web technology, e.g. OWL-S, to create a powerful and practical system that is able to dynamically and automatically integrate web services.

Bibliography

[1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 21–37. Springer-Verlag, 1998.

[2] J. P. A. Almeida, M. Wegdam, M. V. Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for corba. In *Proceedings of the third International Symposium on Distributed Objects and Apllication*, pages 197–207, 2001.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf, May 2003.

[4] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, S. Pogliani D. Orchard, and K. Riemer et al (Editors). Web service choreography interface (wsci) 1.0. http://www.w3.org/TR/wsci/, August 2002. W3c Note.

[5] Daniel Austin, Abbie Barbir, Christopher Ferris, and Sharad Garg (Editors). Web service architecture requirements. http://www.w3.org/TR/wsa-reqs/, February 2004. World Wide Web Consortium (W3C) Working Group Note.

[6] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, and Venkatesh Chopella et al (Editors). Web services conversation language (wscl) 1.0. http://www.w3.org/TR/2002/NOTE-wscl10-20020314/, March 2002. W3C Note.

[7] L. Bellissard, N. De Palma, and M. Riveill. Dynamic reconfiguration of agent-based applications. In *Proceedings of the ACM European SIGOPS Workshop*, Sintra, September 1998.

[8] K. Beznosov. Object security attributes: Enabling application-specific access control in middleware. In *Proceedings of the DOA/CoopLS/ODBASE 2002*, volume 2519 of *LNCS*, pages 693–710, 2002.

[9] Paul V. Biron, Kaiser Permanente, and Ashok Malhotra (Editors). Xml schema part 2: Datatypes (second edition). http://www.w3.org/TR/xmlschema-2/, October 2004. W3C Recommendation 28.

[10] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the Middleware'98*, pages 191–206. Springer-Verlag, September 1998.

[11] D. Booth and C. K. Liu (Editors). Web services description language (wsdl) version 2.0 part 0: Primer. http://www.w3.org/TR/wsdl20-primer/, August 2005. World Wide Web Consortium (W3C) Working Draft.

[12] David Booth, Hugo Haas, and Francis McCabe et al (Editors). Web services architecture. http://www.w3.org/TR/ws-arch/, February 2004. World Wide Web Consortium (W3C) Working Group Note.

[13] D. Box. *Essential COM*. Addison-Wesley Publishing Company, 1998.

[14] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley Press, July 1996. ISBN: 0-471-95869-7.

[15] J. Cao, A. T. S. Chan, Y. Sun, and K. Zhang. Dynamic configuration management in graph-oriented distributed programming environment. *Science of Computer Programming*, 48(1):43–65, July 2003.

[16] F. Casati, S. IInicki, L. Jin, V. Krishnamoorthy, and M. Shan. Adaptive and dynamic service composition in eflow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, volume 1789 of *LNCS*, pages 13–31, 2000.

[17] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. eflow: A platform for developing and managing composite e-services. In *Proceedings of the Academia Industry Working Conference on Research Challenges (AIWoRC)*, pages 341–348, 2000.

[18] M. Castaldi, A. Carzaniga, P. Inverardi, and A. L. Wolf. A lightweight infrastructure for reconfiguring applications. In *Proceedings of the SCM 2003*, volume 2649 of *LNCS*, pages 231–244, 2003.

[19] W. Cazzola. *Communications-Oriented Reflection: a Way to Open Up the RMI Mechanism*. PhD thesis, Department of Computer Science, Universita degli Studi di Milano, Milan, November 2000.

[20] D. W. Chadwick and A. Otenko. The permis x.509 role based privilege management infrastructure. *Future Generation Computer Systems*, 19(2):277–289, February 2003.

[21] M. A. Cibran and B. Verheecke. Dynamic business rules for web service composition. In *Proceedings of the International Workshop on Dynamic Aspects workshop*, Chicago, USA, March 2005.

[22] Microsoft Cooperation. The .net framework developer center. http://msdn.microsoft.com/netframework/.

[23] P. C. David and T. Ledoux. An infrastructure for adaptable middleware. In *Proceedings of On the Move to Meaningful Internet Systems - DOA/CoopIS/ODBASE 2002*, volume 2519 of *LNCS*, pages 773–790, 2002.

[24] Guy. Eddon and Henry. Eddon. *Inside Distributed COM*. Microsoft Press, 1998. ISBN: 157231849X.

[25] M. Endler. A language for generic dynamic configuration of distributed programs. In *Proceedings of the 12th Brazilian Symposium of Computer Networks -SBRC*, pages 175–187, Curitiba, 1994.

[26] M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 68–79, 1992.

[27] David C. Fallside and Priscilla Walmsley (Editors). Xml schema part 0: Primer (second edition). http://www.w3.org/TR/xmlschema-0/, October 2004. W3C Recommendation.

[28] S. Farrell and R.Housley. An internet attribute certificate profile for authorization. http://www.ietf.org/rfc/rfc3281.txt, April 2002. Network Working Group RFC 3281.

[29] Syntegra Federal. Secure access middleware - information assurance for the us intelligence community. White paper, Syntegra Federal, 2003.

[30] S. N. Foley, T. B. Quillinan, and J. P. Morrison. Secure component distribution using webcom. In *Proceedings of the 17th International Conference on Information Security*, Cairo, Egypt, May 2002.

[31] S. N. Foley, T. B. Quillinan, B. Mulcahy, M. O'Connor, and J. P. Morrison. A framework for heterogeneous middleware security. In *Proceedings of the 13th Heterogeneous Computing Workshop*, Santa Fe, New Mexico, USA, April 2004.

[32] Apache Software Foundation. Apache axis project. http://ws.apache.org/axis/.

[33] Apache Software Foundation. Apache wsif project. http://ws.apache.org/wsif/.

[34] Michael Franz. Dynamic linking of software components. *IEEE Computer*, 30(3):74–81, 1997.

[35] K. Fujii and T. Suda. Dynamic service composition using semantic information. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, November 2004.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley Professional Computing Series, October 1994. ISBN: 0201633612.

[37] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of the CASCON97*, pages 169–183, Toronto, 1997.

[38] K. M. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 62–69. IEEE Computer Society Press, 1996.

[39] Object Management Group. Authorization token layer acquisition service (atlas) sepcification - version 1.0. http://www.omg.org/technology/documents/corba_spec_catalog.htm, October 2002.

[40] Object Management Group. Common object request broker architecture: Core specification version 3.0.2. http://www.omg.org/technology/documents/corba_spec_catalog.htm, December 2002.

[41] Object Management Group. Security service specification version 1.8. http://www.omg.org/technology/documents/corba_spec_catalog.htm, March 2002.

[42] David Hollingsworth. Workflow management coalition the workflow reference model. http://www.wfmc.org/standards/docs/tc003v11.pdf. The Workflow Management Coalition Specification Document Number TC00-1003.

[43] R. Housley, W. Ford, W. Folk, and D. Solo. Internet x.509 public key infrastructure certificate and crl profile. http://www.ietf.org/rfc/rfc2459.txt, January 1999. Network Working Group Request for Comments: 2459.

[44] W. Huang, U. Roth, and Ch. Meinel. Improvement to the smart data server with soap. In *Advances in Communications and Software Technologies*, WSEAS Electrical and Computer Engineering Series, pages 107–111, 2002.

[45] W. Huang, U. Roth, and Ch. Meinel. A flexible middleware platform with piped workflow. In *Proceedings of On The Move to Meaningful Internet Systems 2003: OTM'03 Workshops*, LNCS 2889, pages 950–959. Springer-Verlag, 2003.

[46] IAIK. Iaik jce library. http://jce.iaik.tugraz.at/.

[47] JavaSoft. Java core reflection: Api and specification. http://java.sun.com/j2se/1.4.2/docs/guide/reflection/index.html, September 1996.

[48] M. H. Kang, J. N. Froscher, and I. S. Moskowitz. An architecture of multi-level secure interoperability. In *Proceedings of 13th Annual Computer Security Applications Conference*, pages 194–204, 1997.

[49] Graham Klyne, Jeremy J. Carroll, , and Brian McBride (Editors). Resource description framework (rdf) concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, February 2004. W3C Recommendation.

[50] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhaes, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflevtive orb. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000.

[51] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[52] J. Kramer and J. Magee. Dynmic structure in software architectures. In *Proceedings of the Fourth ACM Sigsoft Sympsium On Foundations of Software Engineering*, California, USA, October 1996.

[53] J. Kramer and J. Magee. Analysing dynamic change in software architectures - a case study. In *Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems*, Annapolis, May 1998.

[54] F. Leyman. Web services flow language version 1.0. http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001. IBM Software Group.

[55] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[56] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the OOPSLA'87*, volume 22, pages 147–155. ACM Press, 1987.

[57] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5 European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer-Verlag, September 1995.

[58] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. Mcllraith, S. Narayanan, and M. Paolucci et al. Owl-s: Semantic markup for web services. http://www.daml.org/services/owl-s/1.1/, November 2004.

[59] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transaction on Software Engineering*, 26(1):70–93, 2000.

[60] Nenad Medvidovic. Adls and dynamic architecture changes. In *Proceedings of the SIGSOFT Workshop*, San Francisco, USA, 1996.

[61] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996. ISBN:0849385237.

[62] Sun Microsystems. Enterprise javabeans specification, version 2.1. http://java.sun.com/products/ejb/docs.html, November 2003.

[63] Sun Microsystems. Java remote method invocation. http://java.sun.com/products/jdk/rmi/reference/docs/index.html, 2003.

[64] Sun Microsystems. Java2 platform, enterprise edition specification v1.4. http://java.sun.com/j2ee/j2ee-1_4-pfd2-spec.pdf, November 2003.

[65] N. Mitra. Simple object access protocol (soap) version 1.2 part 0: Primer. http://www.w3.org/TR/soap12-part0/, June 2003. World Wide Web Consortium (W3C) Recommendation.

[66] K. Moazami-Goudarzi. *Consistency-Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 180 Queen's Gate, London SW7 2BZ, UK, 1997.

[67] A. Mukhija and M. Glinz. Casa - a contract based adaptive software architecture framework. In *Proceedings of the 3rd IEEE Workshop on Applications and Services in Wireless Network*, pages 275–286, Bern, July 2003.

[68] A. Mukhija and M. Glinz. A framework for dynamically adaptive applications in a self-organized mobile network environment. In *Proceedings of the ICDCS 2004 Workshop on Distributed Auto-adaptive and Reconfigurable Systems*, 2004.

[69] D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. Total-order planning with partially ordered subtasks. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 425–C430, Morgan Kaufmann, San Francisco, 2001.

[70] S. Nenadic, N. Zhang, and S. Barton. Fides - a middleware e-commerce security solution. In *Proceedings of the 3rd European Conference on Information Warfare and Security*, pages 295–304, 2004.

[71] OASIS. Introduction to uddi: Important features and functional concepts. http://uddi.org/pubs/uddi-tech-wp.pdf, October 2004.

[72] OASIS. Universal description, discovery and integration (uddi) version 3.0.2. http://uddi.org/pubs/uddi-v3.0.2-20041019.pdf, October 2004. UDDI Spec Technical Committee Draft.

[73] P. Oreizy, M. M. Gorlick, T. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[74] N. De Palma, L. Bellissard, and M. Riveill. Dynamic reconfiguration of agent-based applications. In *Proceedings of the Third European Research Seminar on Advances in Distributed Systems*, Madeira Island - Portugal, April 1999.

[75] The Community Open Source Project. Java implementation of corba - openorb. http://openorb.sourceforge.net/.

[76] Allen R. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[77] A. Ramdane-Cherif and N. Levy. An approach for dynamic reconfigurable software architecture. In *Proceedings of the Integrated Design and Process Technology*, 2002.

[78] A. Rasche and A. Polze. Configuration and dynamic reconfiguration of component-based applications with microsoft .net. In *Proceedings of the International Symposium on Object-oriented Real-time distributed Computing*, Hakodate, Japan, 2003.

[79] A. Rasche and A. Polze. Dynamic reconfiguration of component-based real-time software. In *Proceedings of the Workshop on Object-oriented Dependable Real-time Systems*, Sedona, Arizona, USA, February 2005.

[80] D. Riehle and H. Zuellighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):2–13, 1996.

[81] U. Roth, E. G. Haffner, T. Engel, and Ch. Meinel. The smart data server - a new kind of middle tier. In *Proceedings of the IASTED International Conference Internet and Multimedia Systems and Applications*, pages 361–365, 1999.

[82] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[83] S. K. Shivastava and S. M. Wheater. Architectural support for dynamic reconfiguration of large scale distributed applications. In *Proceedings of the 4 th Int. Conf. On Configurable Distributed Systems*, pages 10–17. IEEE Computer Society, May 1998.

[84] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, December 1997. ISBN: 0201178885.

[85] ILEX Development Team. Windows version implementation of openldap. http://www.ilex.fr/openldap/.

[86] Thuan Thai and Hoang Q. Lam. *.NET Framework Essentials (First Edition)*. O' Reilly Press, 2001. ISBN: 0596001657.

[87] S. Thatte. Xlang: Web service for business process design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001. Microsoft Corporation.

[88] W. M. P. van. der Aalst and A. H. M. ter Hofstede. Yawl: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[89] W. M. P. van. der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.

[90] Wil. van. der Aalst. Workflow pattern web site. http://www.workflowpatterns.com/.

[91] M. Vukovic and P. Robinson. Adaptive, planning-based, web service composition for context awareness. In *Proceedings of the International Conference on Pervasive Computing*, Vienna, April 2004.

[92] M. Wermelinger. A hierarchic architecture model for dynamic reconfiguration. In *Proceedings of the second International Workshop on Software Engineering for Parallel and Distributed Systems*, 1997.

[93] WfMC. The workflow management coalition official website. http://www.wfmc.org/.

[94] D. Wu, E. Sirin, J. Hendler, D. Nau, and B. Parsia. Automatic web services composition using shop2. In *Proceedings of the Workshop on Planning for Web Services*, Trento, Italy, June 2003.

[95] L. Zeng, B. Benatallah, H. Lei, A. Ngu, D. Flaxer, and H. Chang. Flexible composition of enterprise web services. *Electronic Markets*, 13(2):141–152, June 2003.

Appendices

Appendix A

Meta-Definition of RBW Schema

Technology of XML Schema is employed to describe the meta-definition of RBW Schema which is a XML based configuration language to specify how component is wrapped and how components interact in RBW.

```
<?xml version="1.0" encoding="UTF-8"?> <rbw:schema
    xmlns="http://www.hpi.uni-potsdam.de/TI/Projeckte/RBW"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:rbw="http://www.hpi.uni-potsdam.de/TI/Projeckte/RBWSchema"
    targetNamespace="http://www.hpi.uni-potsdam.de/TI/Projeckte/RBW"
    elementFormDefault="qualified">

    <rbw:simpleType name="description">
        <xsd:extension base="xsd:string"/>
    </rbw:simpleType>
    <rbw:simpleType name="classType">
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[A-Za-z_][A-Za-z1-9_\.]*[A-Za-z1-9_]"/>
        </xsd:restriction>
    </rbw:simpleType>
    <rbw:complexType name="parameter">
        <xsd:extension based="xsd:any">
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute ref="classType" />
            <xsd:attribute name="isArray" type="xsd:boolean"/>
        </xsd:extension>
    </rbw:complexType>
    <rbw:simpleType name="ioDirection">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="in"/>
            <xsd:enumeration value="out"/>
        </xsd:restriction>
    </rbw:simpleType>
```

```
<rbw:simpleType name="portType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="operation"/>
        <xsd:enumeration value="stream"/>
    </xsd:restriction>
</rbw:simpleType>
<rbw:simpleType name="usage">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="optional"/>
        <xsd:enumeration value="required"/>
    </xsd:restriction>
</rbw:simpleType>
<rbw:complexType name="property">
    <xsd:extension based="xsd:any">
        <xsd:attribute name="name" type="xsd:string"/>
    </xsd:extension>
</rbw:complexType>
<rbw:simpleType name="identifier">
    <xsd:restriction based="xsd:string">
        <xsd:pattern value=".+#.+"/>
    </xsd:restriction>
</rbw:simpleType>
<rbw:complexType name="port">
    <xsd:sequence>
        <xsd:element ref="description" minOccurs="0"
                                       maxOccurs="1"/>
        <xsd:element ref="ioDirection"/>
        <xsd:element ref="portType"/>
        <xsd:element ref="usage"/>
    </xsd:sequence>
    <xsd:attribute ref="identifier"/>
    <xsd:attribute ref="classType"/>
    <xsd:attribute name="defaultValue" type="xsd:any"
                   minOccurs="0" maxOccurs="1"/>
</rbw:complexType>
<rbw:complexType name="component">
    <xsd:sequence>
        <xsd:element ref="description" minOccurs="0"
                                       maxOccurs="1"/>
        <xsd:element ref="parameter" minOccurs="0"
                                     maxOccurs="unbounded"/>
        <xsd:element ref="property" minOccurs="0"
                                    maxOccurs="unbounded"/>
        <xsd:element ref="port" minOccurs="2"
                                maxOccurs="unbounded"/>
```

```
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute ref="classType"/>
</rbw:complexType>
<rbw:simpleType name="linkType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="ANDLink"/>
        <xsd:enumeration value="ORLink"/>
        <xsd:enumeration value="XORLink"/>
        <xsd:enumeration value="MapLink"/>
    </xsd:restriction>
</rbw:simpleType>
<rbw:complexType name="controlLink">
    <xsd:sequence>
        <xsd:element ref="port" minOccurs="2"
                                maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="linkType"/>
</rbw:complexType>
<rbw:complexType name="namedObject">
    <xsd:sequence>
        <xsd.element ref="description" minOccurs="0"
                                       maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute ref="identifier"/>
    <xsd:attribute ref="classType"/>
</rbw:complexType>
<rbw:complexType name="inputs">
    <xsd:sequence>
        <xsd:element ref="description" minOccurs="0"
                                       maxOccurs="1"/>
        <xsd:element ref="namedObject" minOccurs="1"
                                       maxOccurs="unbounded"/>
    </xsd:sequence>
</rbw:complexType>
<rbw:complexType name="outputs">
    <xsd:sequence>
        <xsd:element ref="description" minOccurs="0"
                                       maxOccurs="1"/>
        <xsd:element ref="namedObject" minOccurs="1"
                                       maxOccurs="unbounded"/>
    </xsd:sequence>
</rbw:complexType>
<rbw:simpleType name="import">
    <xsd:extension base="xsd:string"/>
```

```
    </rbw:simpleType>
    <rbw:complexType name="link">
        <xsd:sequence>
            <xsd:element name="source" type="identifier"/>
            <xsd:element name="target" type="identifier"/>
        </xsd:sequence>
    </rbw:complexType>
    <rbw:complexType name="connectors">
        <xsd:sequence>
            <xsd:element ref="link" minOccurs="2"
                                    maxOccurs="unbounded"/>
        </xsd:sequence>
    </rbw:complexType>
    <rbw:complexType name="routing">
        <xsd:sequence>
            <xsd:element ref="description" minOccurs="0"
                                           maxOccurs="1"/>
            <xsd:element ref="inputs"/>
            <xsd:element ref="outputs"/>
            <xsd:element ref="import" minOccurs="1"
                                      maxOccurs="unbounded"/>
            <xsd:element ref="linkControl" minOccurs="0"
                                           maxOccurs="unbounded"/>
            <xsd:element ref="connectors"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"/>
    </rbw:complexType>

</rbw:schema>
```

Appendix B

RBW Schema Example for SDS3

This is a configuration example of SDS3 which is specified in RBW Schema. In the configuration example all secure components are employed to enable full security control on the SDS3 applications.

```
<?xml version="1.0" ?>
<!--======================================================== -->
<!--    Smart Data Server version 3.0  XML configuration     -->
<!--======================================================== -->
<project name="RBW_SDS3"
    xmlns="http://www.hpi.uni-potsdam.de/TI/Projekte/RBW"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.hpi.uni-potsdam.de/TI/
                              Projekte/RBW/config/rbw.xsd">
    <description>...</description>
  <!--  ********************************************** -->
  <!--                Available Components              -->
  <!--  **********************************************-->
  <component name="RequestBroker"
             classType="ti.sds.component.requestBroker.RequestBroker">
    <description>...</description>
    <property name="PoolSize">3</property>
    <port identifier="RequestBroker#UserID"
          classType="java.lang.String" defaultValue="guest">
        <description>userID indicate who is using ...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>optional</usage>
    </port>
    <port identifier="RequestBroker#Permission"
          classType="java.lang.Boolean" defaultValue="true">
        <description>indicates whether the ...</description>
```

```
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>optional</usage>
    </port>
    <port identifier="RequestBroker#Target"
          classType="java.lang.String">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="RequestBroker#Operation"
          classType="java.lang.String">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="RequestBroker#Parameters"
          classType="ti.rbw.routing.ParameterStream">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="RequestBroker#Result"
          classType="java.lang.Object">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="RequestBroker#Context"
          classType="java.lang.String">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
  </component>
  <component name="Authenticator"
     classType="ti.sds.component.authentication.Authenticator">
    <description>...</description>
    <property name="PoolSize">3</property>
    <port identifier="Authenticator#Signature"
```

```
          classType="java.lang.Byte[]">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="Authenticator#UserID"
          classType="java.lang.String">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="Authenticator#Valid"
          classType="java.lang.Boolean">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
  </component>
  <component name="Authorizer"
    classType="ti.sds.component.authorization.Authorizer">
    <description>...</description>
    <property name="PoolSize">3</property>
    <port identifier="Authorizer#UserID"
          classType="java.lang.String">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="Authorizer#Roles"
          classType="ti.security.pkc.Role[]">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
  </component>
  <component name="AccessController"
     classType="ti.sds.component.accessControl.AccessController">
    <description>...</description>
    <property name="PoolSize">3</property>
    <port identifier="AccessController#Valid"
```

```
        classType="java.lang.Boolean" defaultValue="true">
      <description>...</description>
      <ioDirection>in</ioDirection>
      <portType>operation</portType>
      <usage>optional</usage>
  </port>
  <port identifier="AccessController#Target"
        classType="java.lang.String">
      <description>...</description>
      <ioDirection>in</ioDirection>
      <portType>operation</portType>
      <usage>required</usage>
  </port>
  <port identifier="AccessController#Operation"
        classType="java.lang.String">
      <description>...</description>
      <ioDirection>in</ioDirection>
      <portType>operation</portType>
      <usage>required</usage>
  </port>
  <port identifier="AccessController#Roles"
        classType="ti.security.pkc.Role[]">
      <description>...</description>
      <ioDirection>in</ioDirection>
      <portType>operation</portType>
      <usage>required</usage>
  </port>
  <port identifier="AccessController#Permission"
        classType="java.lang.Boolean">
      <description>...</description>
      <ioDirection>out</ioDirection>
      <portType>operation</portType>
      <usage>required</usage>
  </port>
</component>
<component name="AbnormalHandler"
    classType="ti.sds.component.abnormal.AbnormalHandler">
  <description>...</description>
  <property name="PoolSize">3</property>
  <port identifier="AbnormalHandler#Permission"
        classType="java.lang.Boolean" defaultValue="true">
      <description>...</description>
      <ioDirection>in</ioDirection>
      <portType>operation</portType>
      <usage>optional</usage>
```

```
    </port>
    <port identifier="AbnormalHandler#Result"
          classType="java.lang.Object">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="AbnormalHandler#Context"
          classType="java.lang.String">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
  </component>
  <!-*****************************************************-->
  <!--          Available Routings                       -->
  <!--*****************************************************-->
  <routing name="FullControlRouting">
    <description>...</description>
    <inputs>
        <namedValue identifier="Request#Target"
                    classType="java.lang.String">
            <description>..</description></namedValue>
        <namedValue identifier="Request#Operation"
                    classType="java.lang.String">
            <description>...</description></namedValue>
        <namedValue identifier="Request#Parameters"
                    classType="ti.rbw.routing.ParameterStream">
            <description>...</description></namedValue>
        <namedValue identifier="Request#Signature"
                    classType="java.lang.Byte[]">
            <description>...</description></namedValue>
    </inputs>
    <outputs>
        <namedValue identifier="Response#Result"
                    classType="java.lang.Object">
            <description>...</description></namedValue>
        <namedValue identifier="Response#Context"
                    classType="java.lang.String">
            <description>...</description></namedValue>
    </outputs>
    <import>Authenticator</import>
    <import>Authorizer</import>
```

```
<import>AccessController</import>
<import>RequestBroker</import>
<import>AbnormalHandler</import>
<controlLink name="Selector" linkType="AndLink">
    <port identifier="Permission" classType="java.lang.Boolean">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="OutTrue" classType="java.lang.Boolean">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="OutFalse" classType="java.lang.Boolean">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
</controlLink>
<connectors>
  <link>
    <source>Request#Signature</source>
    <target>Authenticator#Signature</target></link>
  <link>
    <source>Authenticator#Valid</source>
    <target>AccessController#Valid</target></link>
  <link>
    <source>Authenticator#UserID</source>
    <target>Authorizer#UserID</target></link>
  <link>
    <source>Authenticator#UserID</source>
    <target>RequestBroker#UserID</target></link>
  <link>
    <source>Authorizer#Roles</source>
    <target>AccessController#Roles</target></link>
  <link>
    <source>Request#Target</source>
    <target>AccessController#Target</target></link>
  <link>
    <source>Request#Operation</source>
    <target>AccessController#Operation</target></link>
```

```
      <link>
        <source>Request#Target</source>
        <target>RequestBroker#Target</target></link>
      <link>
        <source>Request#Operation</source>
        <target>RequestBroker#Operation</target></link>
      <link>
        <source>Request#Parameters</source>
        <target>RequestBroker#Parameters</target></link>
      <link>
        <source>RequestBroker#Result</source>
        <target>Response#Result</target></link>
      <link>
        <source>RequestBroker#Context</source>
        <target>Response#Context</target></link>
      <link>
        <source>AccessController#Permission</source>
        <target>Selector#Permission</target></link>
      <link>
        <source>Selector#OutTrue</source>
        <target>RequestBroker#Permission</target></link>
      <link>
        <source>Selector#OutFalse</source>
        <target>AbnormalHandler#Permission</target></link>
      <link>
        <source>AbnormalHandler#Result</source>
        <target>Response#Result</target></link>
      <link>
        <source>AbnormalHandler#Context</source>
        <target>Response#Context</target></link>
    </connectors>
  </routing>
  <!--************************************************ -->
  <!--      Available Application Services              -->
  <!--************************************************ -->
  <application name="StringHandler"
               classType="ti.sds.servant.sample.StringHandler">
      <description>This is only for test!</description>
  </application>
</project>
```

Appendix C

Policy Example for Access Controller

A policy example for component of Access Controller is give here to show how to enable application-specific security control by configuring policy.

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE SDS3_RBAC_Policy>
<X.509_SDS3_RBAC_Policy OID="1.2.826.0.1.3344810.6.0.19">
    <SubjectPolicy>
        <SubjectDomainSpec ID="Developers">
            <Include LDAPDN="OU=TI,O=HPI,C=DE"/>
        </SubjectDomainSpec>
        <SubjectDomainSpec ID="Utilizer">
            <Include LDAPDN="C=DE"/>
        </SubjectDomainSpec>
    </SubjectPolicy>
    <SOAPolicy>
       <SOASpec ID="PolicyOwner" LDAPDN="CN=SOA,OU=TI,O=HPI,C=DE"/>
    </SOAPolicy>
    <RoleHierarchyPolicy>
        <RoleSpec OID="1.2.826.0.1.3344810.1.1.14" Type="UserRole">
            <SupRole Value="Guest"/>
            <SupRole Value="Clerk"/>
            <SupRole Value="Customer"/>
            <SupRole Value="Manager">
                <SubRole Value="Clerk"/>
                <SubRole Value="Customer"/></SupRole>
            <SupRole Value="Administrator">
                <SubRole Value="Manager"/></SupRole>
        </RoleSpec>
        <RoleSpec OID="1.2.826.0.1.3344810.1.1.15" Type="AuthorRole">
            <SupRole Value="Tester"/>
            <SupRole Value="Programmer"/>
```

```
            <SupRole Value="Designer">
                <SubRole Value="Tester"/>
                <SubRole Value="Programmer"/></SupRole>
            <SupRole Value="Supervisor">
                <SubRole Value="Designer"/></SupRole>
        </RoleSpec>
    </RoleHierarchyPolicy>
    <RoleAssignmentPolicy>
        <RoleAssignment>
            <SubjectDomain ID="Developers"/>
            <RoleList>
                <Role Type="AuthorRole" Value="Tester"/>
                <Role Type="AuthorRole" Value="Programmer"/>
                <Role Type="AuthorRole" Value="Designer"/>
                <Role Type="AuthorRole" Value="Supervisor"/>
            </RoleList>
            <Delegate Depth="0"/>
            <SOA ID="PolicyOwner"/>
            <Validity>
                <Absolute Start="2004-09-21T00:00:00"/>
                <Absolute End="2005-12-21T00:00:00"/></Validity>
        </RoleAssignment>
        <RoleAssignment>
            <SubjectDomain ID="Utilizer"/>
            <RoleList>
                <Role Type="UserRole" Value="Guest"/>
                <Role Type="UserRole" Value="Clerk"/>
                <Role Type="UserRole" Value="Customer"/>
                <Role Type="UserRole" Value="Manager"/>
                <Role Type="UserRole" Value="Administrator"/>
            </RoleList>
            <Delegate Depth="0"/>
            <SOA ID="PolicyOwner"/>
            <Validity>
                <Absolute Start="2004-09-21T00:00:00" />
                <Absolute End="2005-12-21T00:00:00" /></Validity>
        </RoleAssignment>
    </RoleAssignmentPolicy>
    <TargetPolicy>
        <TargetDomainSpec ID="TargetInvocation">
          <Include LDAPDN="CN=Invoker,OU=TI,O=HPI,C=DE" />
        </TargetDomainSpec>
        <TargetDomainSpec ID="TargetConfiguration">
          <Include LDAPDN="CN=Configurator,OU=TI,O=HPI,C=DE"/>
        </TargetDomainSpec>
```

```
    </TargetPolicy>
    <ActionPolicy>
        <Action Args="" Name="invoke" />
        <Action Args="" Name="insert" />
        <Action Args="" Name="remove" />
        <Action Args="" Name="modify" />
    </ActionPolicy>
    <TargetAccessPolicy>
        <TargetAccess>
            <RoleList>
                <Role Type="UserRole" Value="Clerk" />
                <Role Type="UserRole" Value="Customer" />
                <Role Type="AuthorRole" Value="Tester" />
                <Role Type="AuthorRole" Value="Programmer" />
            </RoleList>
            <TargetList>
                <Target Actions="invoke">
                    <TargetDomain ID="TargetInvocation" />
                </Target>
            </TargetList>
        </TargetAccess>
        <TargetAccess>
            <RoleList>
                <Role Type="UserRole" Value="Manager" />
                <Role Type="AuthorRole" Value="Designer" />
            </RoleList>
            <TargetList>
                <Target Actions="invoke">
                    <TargetDomain ID="TargetConfiguration" />
                </Target>
                <Target Actions="insert">
                    <TargetDomain ID="TargetConfiguration" />
                </Target>
                <Target Actions="remove">
                    <TargetDomain ID="TargetConfiguration" />
                </Target>
                <Target Actions="modify">
                    <TargetDomain ID="TargetConfiguration" />
                </Target>
            </TargetList>
        </TargetAccess>
    </TargetAccessPolicy>
</X.509_SDS3_RBAC_Policy>
```

Appendix D

RBW Schema Example for DSC

An example of RBW Schema for web services composition is given to show how to integrate web services in DSC. For better understanding, the name of items *component* and *routing* are respectively changed to *BasicService* and *CompositeService* in the configuration of DSC.

```
<?xml version="1.0" ?> -
<!--============================================================-->
<!--Configuration for Dynamic Services Composer Supported by RBW -->
<!--============================================================-->
<project name="RBW_DSC"
    xmlns="http://www.hpi.uni-potsdam.de/TI/Projekte/RBW"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.hpi.uni-potsdam.de
                            /TI/Projekte/RBW/config/rbw.xsd">
    <description>...</description>

    <!--*********************************************************-->
    <!--               Available Basic Services                  -->
    <!--*********************************************************-->
    <BasicService name="GetCurrencyRate"
                  classType="ti.ws.invoker.RemoteInvoker">
        <description>...</description>
        <parameter name="WSDLDoc" classType="java.lang.String">
                 .\wsdl\CurrencyExchangeService.wsdl</parameter>
        <parameter name="Interface" classType="java.lang.String">
                                    getRateSoap</parameter>
        <parameter name="Operation" classType="java.lang.String">
                                    getRate</parameter>
        <property name="PoolSize">5</property>
        <port identifier="GetCurrencyRate#country1"
                         classType="java.lang.String">
```

```
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="GetCurrencyRate#country2"
                    classType="java.lang.String">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="GetCurrencyRate#Result"
                    classType="java.lang.Float">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
</BasicService>

<BasicService name="EmailSender"
              classType="ti.ws.invoker.RemoteInvoker">
    <description>...</description>
<parameter name="WSDLDoc" classType="java.lang.String">
                  .\wsdl\IEmailService.wsdl</parameter>
<parameter name="Interface" classType="java.lang.String">
                                SendMailSoap</parameter>
<parameter name="Operation" classType="java.lang.String">
                                     SendMail</parameter>
<property name="PoolSize">5</property>
<port identifier="EmailSender#ToAddress"
        classType="java.lang.String">
    <description>...</description>
    <ioDirection>in</ioDirection>
    <portType>operation</portType>
    <usage>required</usage>
</port>
<port identifier="EmailSender#FromAddress"
                  classType="java.lang.String">
    <description>...</description>
    <ioDirection>in</ioDirection>
    <portType>operation</portType>
    <usage>required</usage>
</port>
```

```
<port identifier="EmailSender#ASubject"
                  classType="java.lang.String">
    <description>...</description>
    <ioDirection>in</ioDirection>
    <portType>operation</portType>
    <usage>required</usage>
</port>
<port identifier="EmailSender#MsgBody"
                  classType="java.lang.String">
    <description>...</description>
    <ioDirection>in</ioDirection>
    <portType>operation</portType>
    <usage>required</usage>
</port>
<port identifier="EmailSender#return"
      classType="java.lang.Integer" defaultValue="0">
    <description>...</description>
    <ioDirection>out</ioDirection>
    <portType>operation</portType>
    <usage>optional</usage>
</port>
</BasicService>

<BasicService name="EmailVerifier"
              classType="ti.ws.invoker.RemoteInvoker">
    <description>...</description>
    <parameter name="WSDLDoc" classType="java.lang.String">
                .\wsdl\ValidateEmail.wsdl</parameter>
    <parameter name="Interface" classType="java.lang.String">
                         IsValidEMailSoap</parameter>
    <parameter name="Operation" classType="java.lang.String">
                             IsValidEMail</parameter>
    <property name="PoolSize">3</property>
    <port identifier="EmailVerifier#EmailAddress"
                     classType="java.lang.String">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="EmailVerifier#IsValidEMailResult"
                     classType="java.lang.Boolean">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
```

```
        <usage>required</usage>
    </port>
</BasicService>

<BasicService name="QueryVerifier"
              classType="ti.ws.invoker.LocalInvoker">
    <description>...</description>
    <parameter name="ServiceClass" classType="java.lang.String">
             ti.ws.services.sample.QueryService</parameter>
    <parameter name="Operation"
         classType="java.lang.String">query_verify</parameter>
    <property name="PoolSize">3</property>
    <port identifier="QueryVerifier#IsEmailValid"
            classType="java.lang.Boolean" defaultValue="true">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>optional</usage>
    </port>
    <port identifier="QueryVerifier#EmailAddress"
            classType="java.lang.String">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="QueryVerifier#QueryMessage"
                      classType="java.lang.Double">
        <description>...</description>
        <ioDirection>in</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="QueryVerifier#VerifyResult"
                classType="java.lang.Boolean">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
        <usage>required</usage>
    </port>
    <port identifier="QueryVerifier#QueryResult"
                classType="java.lang.String" defaultValue="NULL">
        <description>...</description>
        <ioDirection>out</ioDirection>
        <portType>operation</portType>
```

```
        <usage>optional</usage>
    </port>
</BasicService>

<!--*****************************************************-->
<!--              Available Composite Services           -->
<!--*****************************************************-->
<CompositeService name="QueryInfoService">
    <description>...</description>
    <inputs>
        <namedValue identifier="Request#fromCurrency"
                classType="java.lang.String">
            <description>...</description></namedValue>
        <namedValue identifier="Request#toCurrency"
                classType="java.lang.String">
            <description>...</description></namedValue>
        <namedValue identifier="Request#subject"
                classType="java.lang.String">
            <description>...</description></namedValue>
        <namedValue identifier="Request#email"
                classType="java.lang.String">
            <description>...</description></namedValue>
    </inputs>
    <outputs>
        <namedValue identifier="Response#return"
                classType="java.lang.Boolean">
            <description>...</description></namedValue>
    </outputs>
    <import>CurrencyConvertor</import>
    <import>EmailSender</import>
    <import>EmailVerifier</import>
    <import>QueryVerifier</import>
    <connectors>
        <link><source>Request#fromCurrency</source>
            <target>CurrencyConvertor#fromCurrency</target></link>
        <link><source>Request#toCurrency</source>
            <target>CurrencyConvertor#toCurrency</target></link>
        <link><source>Request#email</source>
            <target>QueryVerifier#EmailAddress</target></link>
        <link><source>CurrencyConvertor#rate</source>
            <target>QueryVerifier#QueryMessage</target></link>
        <link><source>Request#email</source>
            <target>EmailSender#ToAddress</target></link>
        <link><source>Request#email</source>
            <target>EmailSender#FromAddress</target></link>
```

```
            <link><source>Request#subject</source>
                <target>EmailSender#ASubject</target></link>
            <link><source>QueryVerifier#QueryResult</source>
                <target>EmailSender#MsgBody</target></link>
            <link><source>QueryVerifier#VerifyResult</source>
                <target>Response#return</target></link>
        </connectors>
    </CompositeService>

    <application name="QueryService"
            classType="ti.ws.services.sample.QueryService">
        <description>...</description>
        <function name="query_info"
                  CompositeService="QueryInfoService"/>
    </application>
</project>
```