uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Yongsheng Yang
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Comparative study and implementation methodologies for distributed applications

TITRE DE LA THÈSE / TITLE OF THESIS

G. Bochmann
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

A. El Saddik

D. Petriu

Gary W. Slater
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

Comparative study and implementation methodologies for distributed applications

By

Yongsheng Yang

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the Degree of Master of Applied Science, Electrical Engineering

Ottawa-Carleton Institute for Electrical Engineering
School of Information Technology and Engineering
University of Ottawa

© 2005, Yongsheng Yang

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-11464-9
Our file Notre référence
ISBN: 0-494-11464-9

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

Today's rapid development of Internet demands highly concurrent and distributed systems. In this context, it is interesting to note that correct protocol specifications can be derived from a given service specification of the application.

The main objective of this thesis is to introduce protocol derivation and develop general methodologies for designing distributed systems. Protocol synthesis focuses on generating an error-free protocol specification automatically from a higher-level service specification and it is a very important step in the protocol development process. Petri net is a very powerful model for describing and analyzing distributed, concurrent and asynchronous systems; therefore, we use it in protocol synthesis. Furthermore, we review SOAP (Simple Object Access Protocol), Java RMI (Remote Method Invocation) and J2EE (Java 2 Platform, Enterprise Edition) technologies and discuss their usages in decentralized, distributed applications.

This thesis gives an overview of protocol engineering background knowledge, presents several protocol synthesis methods using Petri nets, gives general methodologies of distributed system implementation and presents a distribution prototype: An example service specification is used to derive a corresponding abstract protocol specification, which is implemented with three different approaches: SOAP, Java RMI and J2EE. Finally, the performance of these different implementation approaches is compared.

Keywords: Petri nets, Protocol Synthesis, SOAP, Java RMI, J2EE, EJB

Acknowledgements

The evolution of this thesis was not a solitary effort. Many people assisted me in both the design of the project and the writing of the thesis.

I would like to express my sincere appreciation to my supervisor, Dr. Gregor v. Bochmann, for his work ethics, rich knowledge, encouragement, and patience throughout my research.

I would like to thank my colleagues at the AIAS lab, especially Jun Chen, Jinzhi Xia, Cheng Peng, Mazen Khair and Abdelilah Maach for their valuable comments and interesting discussions.

Finally, I would like to thank my parents and my wife who support me through many difficulties.

Table of Contents

Abstract *1*

Acknowledgements *2*

TABLE OF CONTENTS *3*

LIST OF FIGURES *5*

LIST OF TABLES *7*

Chapter 1 Introduction ***8***
1.1 Protocol synthesis in distributed systems *8*
1.2 Objective and contribution of this thesis *10*
1.3 Thesis organization *12*

Chapter 2 Background and motivation ***13***
2.1 Overview of protocol engineering *13*
2.1.1 Protocol engineering development life cycle *16*
2.2 Overview of Petri nets *19*
2.3 Introduction to Coloured Petri nets *26*

Chapter 3 Protocol synthesis using Petri nets ***28***
3.1 Overview of protocol synthesis *28*
3.2 Petri nets based protocol derivation *33*
3.2.1 Service decomposition for distributed system design *33*
3.2.2 Protocol derivation example *35*

Chapter 4 Distributed Petri net implementation methodologies ***42***
4.1 General methodologies of implementation *42*
4.2 Local Petri nets control flow analysis *46*
4.2.1 The relationships between places and transitions in Petri nets *46*
4.2.2 Object-oriented modelling of Petri nets *50*
4.2.3 Concurrency in Petri nets *51*
4.2.4 Distributed implementation of Petri nets *54*
4.2.3 Layer pattern *57*
4.3 Transaction management *58*
4.3.1 Distributed transaction concept *58*
4.3.2 Principles of two phase commit protocol implementation *64*
4.3.3 Transaction recovery *65*

Chapter 5 Protocol design and implementation issues ***69***
5.1 Overview of issues *69*
5.2 Message coding and object serialization *70*
5.2.1 Introduction to Java object XML serialization *73*

5.3 Introduction to RPC-based technologies......78
5.3.1 Introduction to SOAP......78
5.3.2 Introduction to Apache AXIS......81
5.3.3 Introduction to Java RMI......82
5.4 Introduction to J2EE......85
5.4.1 Introduction to EJB......88
5.4.2 Introduction to JMS......91
5.5 Introduction to Servlets......93
5.6 Introduction to code generation......93
5.7 Conclusion......94

Chapter 6 Example implementations and comparison......96
6.1 Java RMI based approach......96
6.1.1 Define the remote interface of each processing node......96
6.1.2 Develop the remote object by implementing the remote interface......97
6.1.3 Define the database access layer......97
6.1.4 System testing......98
6.2 SOAP based approach......100
6.2.1 Define the remote interface of each processing node......101
6.2.2 Define and deploy services......101
6.2.3 Define the database access layer......102
6.2.4 System testing......102
6.3 J2EE based approach......104
6.3.1 Define the remote interface of each processing node......104
6.3.2 Define and deploy services on each processing node......104
6.3.3 Define the database access layer......107
6.3.4 System testing......107
6.4 Performance comparison......107

Chapter 7 Conclusions and future work......111
7.1 Conclusions......111
7.2 Future work......112

Reference......113

Appendix A: MySQL database setup......119
Appendix B: Setup AXIS running environment......121
Appendix C: Setup Java RMI running environment and running instructions......123
Appendix D: Setup J2EE running environment and running instructions......124
Appendix E: Setup MySQL in JBOSS......131
Appendix F: SOAP call example......132
Appendix G: Java class diagrams......134
Appendix H: Message length comparison......143

List of Figures

Chapter 2

Figure 2.1: OSI architecture....14
Figure 2.2: SAP and communication service....16
Figure 2.3: Protocol engineering development life cycle....17
Figure 2.4: (a) service specification (b) protocol specification....19
Figure 2.5: Basic symbol in Petri nets....20
Figure 2.6: Petri net example....21
Figure 2.7: Simple Petri net structure....22
Figure 2.8: Initial state....23
Figure 2.9: T is enabled....23
Figure 2.10: T firing ($t \rightarrow 0$)....23
Figure 2.11: End of the firing....24

Chapter 3

Figure 3.1: Views of service and protocol....28
Figure 3.2: Replacement of transition T1....34
Figure 3.3: Service specification example....36
Figure 3.4: Derived service specification....38
Figure 3.5: NodeA....40
Figure 3.6: NodeB....41
Figure 3.7: NodeC....41

Chapter 4

Figure 4.1: Decompose Petri nets....43
Figure 4.2: Sequential relation....47
Figure 4.3: Concurrency relation....48
Figure 4.4: Conflict relation....49
Figure 4.5: Synchronization relation....49
Figure 4.6: Confusion relation....50
Figure 4.7: Transaction states....59
Figure 4.8: Two-phase commit protocol....63
Figure 4.9: First phase....64
Figure 4.10: Second phase....65

Chapter 5

Figure 5.1: Java RMI structure....83
Figure 5.2: Java RMI calling procedure....85
Figure 5.3: 3-tier framework....86
Figure 5.4: J2EE three-tier architecture....87

Figure 5.5: JMS publish-and-subscribe model..91
Figure 5.6: JMS point-to-point model..92
Figure 5.7: Code generation workflow..94

Chapter 6

Figure 6.1: NodeA local transaction records..99
Figure 6.2: NodeB local transaction records..99
Figure 6.3: NodeC local transaction records..100
Figure 6.4: NodeA local transaction records..102
Figure 6.5: NodeB local transaction records..103
Figure 6.6: NodeC local transaction records..103
Figure 6.7: AXIS, Java RMI and J2EE performance compared............................110

List of Tables

Chapter 4

Table 4.1: layers pattern in project..............................58

Chapter 5

Table 5.1: mapping between Java Type and XSD..............................75

Chapter 6

Table 6.1: Hardware configuration..............................108
Table 6.2: AXIS running results..............................108
Table 6.3: Java RMI running results..............................109
Table 6.4: J2EE running results..............................109

CHAPTER 1 Introduction

1.1 Protocol synthesis in distributed systems

A distributed system contains a collection of autonomous computers, connected through network and distribution middleware, which enables computers to coordinate their activities and to share the resource of the system, so that users perceive the system as a single, integrated computing facility [36].

Distributed systems offer the ability for sharing and integrating different resources, such as computers, database systems and other specialized devices. These resources are distributed and owned by different components.

They have the following key characteristics:

1. Resource sharing: resources provided by a computer, which is a member of a distributed system, can be shared by clients and other members of the system via a network.
2. Openness: data access and other services are available through the network for other machines.
3. Concurrency: multiple services for different clients can be provided at the same time. All concurrent access must be synchronized to avoid problems.
4. Scalability: operate effectively at many different scales, ranging from a small intranet to the Internet – flexible to grow in size.
5. Fault tolerance: Any process, computer or network may fail independently of the others. The system can handle errors that occurred.

6. Transparency: certain parts of the distributed system are invisible to developers, so they can only focus on the design of particular application. The idea is to hide all unnecessary details from users.

Due to the big progress of high-speed networks and computer, more and more systems have been implemented as distributed systems. These systems send and receive messages through networks and the control flow is very complex indeed. Designing a highly reliable protocol is still a very complex task. The protocol implementation is the process that takes the protocol specification and develops the protocol software modules.

In order to provide desired services, a communication protocol includes of sets of rules to control the orderly exchange messages among different nodes. Reliable computer communication protocols play a very important role for the communication services in distributed systems. It handles different issues related to distributed systems such as fault tolerance, concurrency, synchronization and communication. There are three properties that must be guaranteed in a protocol design:

- Safety: it ensures that the protocol never enters an undesirable state.
- Liveness: it ensures that the protocol will eventually enter a desirable state.
- Responsiveness: it ensures that protocol has the following features: (1) timeliness, that is, it respects the response time requirements imposed by the service specification; (2) fault-tolerance, that is, the possibility of recovering in the case of failures.

But designing protocols is not an easy thing due to the complexity and other requirement constraints. The protocol design methodologies commonly used in industry are rather informal, because the methodologies usually involve a sequence of trial-and-error steps

that terminate when the designers feel a certain level of confidence in their design. But if we do not use formal techniques, at the end of the design process there is no guarantee that the protocol specification and its implementation will satisfy or reach the initial specification requirements. Let us look at a formal design methodology for the protocol design: protocol synthesis.

Protocol synthesis is an approach to formalize and automate the process of designing communication protocols. Protocol synthesis tries to solve the problem of deriving a set of specifications for the protocol entities from a given service specification of a distributed system.

There are lots of different approaches to derive protocol specification from service specification. We will discuss them in Chapter 2.

1.2 Objective and contribution of this thesis

The objective of the thesis is:

- Develop general methodologies for designing distributed Petri net-based systems.
- Implement highly concurrent, distributed transactions among different nodes.
- Implement distributed Petri-net-based protocol specifications with three different Java approaches.
- Compare the performances of these different approaches.

The contribution of the thesis is as follows:

A new protocol synthesis method was proposed in [40]. The method decomposes each transition of service specification into a set of communicating sub-Petri-nets running on N protocol entities. The method is very useful and can guarantee the liveness and correctness of the generated protocol specification. The goal of the thesis is to develop

general methodologies for designing distributed systems, implement the distributed Petri nets-based protocol specification example with three different approaches: Java RMI, SOAP and J2EE. The general methodologies are developed to handle highly concurrent distributed control flow and transactions of protocol specifications. We first use these methodologies to implement the control flow of the abstract protocol specifications in a single program and the running results indicate the methodologies work correctly. Then we distribute the protocol specification over different nodes, and apply these implementation methodologies on these distributed nodes. The running results prove the solution is correct.

We believe the research is valuable, because:

1. General methodologies for the implementation of Petri net-based distributed system were given. These methodologies can correctly handle highly concurrent distributed control flow.
2. We implemented a distributed Petri nets-based protocol specification example using different approaches: SOAP, Java RMI, J2EE. The performance of these different approaches was compared.
3. We implemented highly concurrent, distributed transactions using Java multiple threads and SQL. The two-phase commit protocol was used to guarantee the ACID [36] properties.

1.3 Thesis Organization

The thesis is organized in the following way: Chapter 2 gives the introduction of background and motivation. In the first part, we will give an overview of protocol engineering, and then we will introduce the concepts of Petri nets then introduce the Coloured Petri nets. In Chapter 3, we will talk about protocol synthesis, especially with a Petri nets based approach. In Chapter 4, will talk about the general methodologies used in distributed Petri nets implementation, first we will discuss local Petri nets control flow analysis, then discuss the distributed Petri nets control flow implementation design. In Chapter 5, we will discuss several Java technologies for distributed computing. These technologies include XML, SOAP, RMI, J2EE, JMS, Servlets and distributed transactions. In Chapter 6, we will discuss the protocol implementation and different approaches based on Java RMI, SOAP and J2EE. Performance issues are also discussed. In Chapter 7, we give the conclusions.

CHAPTER 2 Background and motivation

2.1 Overview of protocol engineering

Distributed applications are complicated because they typically contain a lot of modules interacting and communicating with one another through a network.

The interoperability between processes running on separate computers is very important. Interoperability means that distinct processes, each running on its own processor, cooperate in accomplishing some goal. They cooperate by exchanging information, by notifying each other as various tasks are completed, by requesting services from and granting services to each other. These cooperative operations depend on a common understanding of how these interactions will proceed. The processes must agree on the format and the meaning of the messages exchanged, and under what conditions the exchange will take place. The agreement about what messages will be sent and expected, under what conditions, in what format, and with what meaning is called a protocol.

For instance, the Open Systems Interconnection (OSI) reference model introduces seven layers of protocols to allow heterogeneous cooperation. The OSI architecture [1] is shown in Figure 2.1. The dotted arrows represent logical communication while the solid arrows represent physical communication.

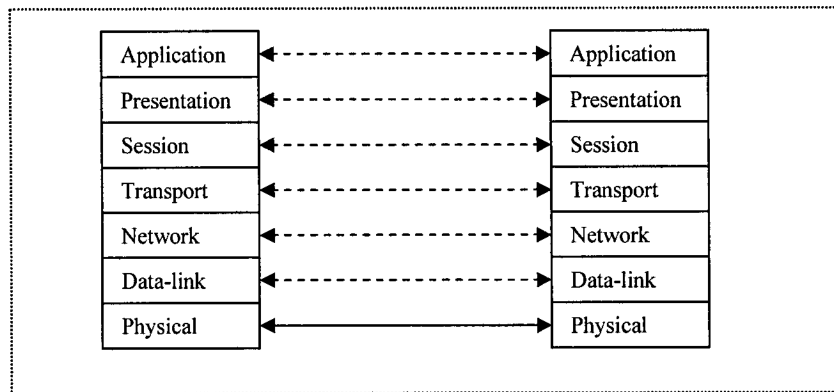


Figure 2.1: OSI architecture

The seven layers of the OSI Model provide a framework for developing standards that permit open systems interconnection. In this architecture, each layer must provide services to the layer above it and can use the services of the layer below it.

There are some important concepts in protocol engineering:

- Service specification: the specification that describes how a protocol layer provides network services to users or protocol modules in upper layer.
- Protocol specification: the specification that describes the message format, exchange sequences of protocol messages for each protocol messages, thus realizing the service specification.
- Protocol synthesis: the process that generates error-free protocol specification from service specification.
- Protocol validation: the process that verifies if generated protocol specification can realize the service specification.
- Conformance testing: the process that generates several test cases based on protocol specification to test the protocol implementation.

Depending on the level of abstraction, a distributed system can be described at these levels:

- Service specification level: at this level, the system can be viewed as a service provider that offers some communication services. The services of a protocol can be accessed at its Service Access Points (SAPs) by users. At the service specification level, the properties of the services should be specified and it will give the possible scenario where different actions may occur at different SAPs. Usually, the service can be specified by some sort of finite state machine.
- Protocol specification level: at this level, the protocol is modeled as several communicating processes called protocol entities. Each protocol entity represents a location or a process that is distributed from the other entities and communicates with other entities by message passing. So it is very important to design a proper message format to exchange information among entities. At this level, we need protocol validation to make sure the protocol specification meets the service specification.

A user can use the interface of the service access point to access the communication service. The specifications of the communication service and protocol have global relevance. The interface properties can be different for different service access points, but they have to be consistent with local properties of a SAP as defined by the service specification. Communication service and service access points are shown in Figure 2.2.

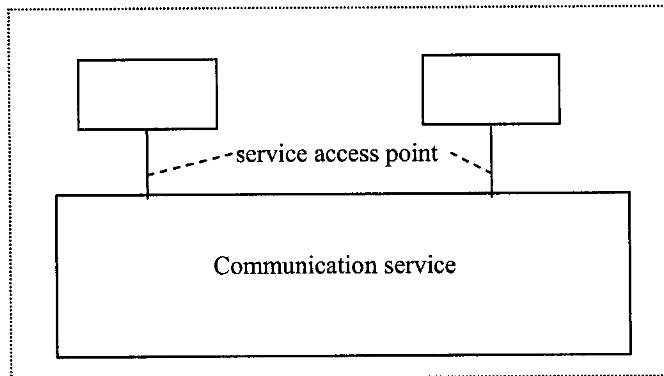


Figure 2.2: SAP and communication service

Protocol entity (PE) provides communication service to users; meanwhile, it is the user of the communication service below. The primitive interactions between a protocol entity and the user are called service primitives. Service primitives are usually invoked at a SAP (between communication service and user).

Protocol Data Units (PDUs) are the interactions exchanged between protocol entities. But the PDUs are not exchanged directly with peers but by coding the PDUs and parameters in user data exchanged through the underlying communication service.

2.1.1 Protocol engineering development life cycle

Usually, the protocol engineering development life cycle includes the following phases:

- Requirement analysis: find the problem to be solved and get the service specification.
- Design phase: develop a detailed system specification.
- System implementation.
- System testing.

The functional specifications of the system including service specification and protocol specification are created from an informal description during the requirement analysis;

also a validation activity is needed to make sure that the protocol specification correctly relates to the service specification. Conformance tests are used to test the protocol implementation during the implementation phase.

The detailed steps are shown in the following Figure 2.3.

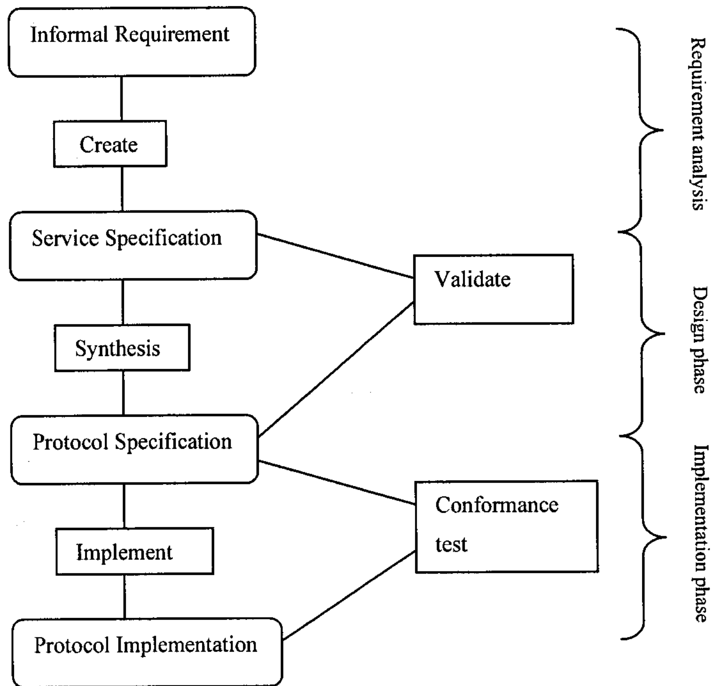


Figure 2.3: Protocol engineering development life cycle

Specifications are written in natural or formal language and they cannot be created automatically. But some computer tools can be useful for the creation.

Since protocol synthesis is a very complex process and is the basis for the protocol specification, we will discuss it in the next chapter.

The following formal methods can be used for specification validation:

- Static analysis: static analysis is very useful for finding clerical errors. There are some tools can check context-free syntax, scope rules and other semantic conditions.
- Dynamic analysis: dynamic analysis considers some kind of execution of the specified system. It can be classified into exhaustive and simulation methods. The exhaustive method will consider all the possibilities during system execution. The simulation method validates only several selected paths among all the possible executions.

After we get the protocol specification, we can start the protocol implementation. The developing process includes several steps of refinement and several tools can be used to generate code partly automatically.

In order to validate the protocol implementation, test cases are used. Test cases should include major aspects of behaviour to check the correctness, performance and robustness of the implementation. Each test case defines the input applied to the system and test result analysis is used to verify whether the trace of interactions observed during the test satisfies the requirement of the protocol specification.

Due to factors such as hardware problems, synchronous and asynchronous messages, the protocol specifications usually are much more complex than the service specifications.

Let us look at a simple example that shows how a protocol specification relates to its service specification (see Figure 2.4). The idea is that only after process A has updated the value of v, process B can use variable v. The two actions: +IN (receive message) and –IN (send message) are used to model the protocol specification level of the

communication between the process PE_1 and PE_2. The service specification does not show the actual communication between these processes.

During the protocol design phase, some protocol entity modules are generated to send and receive messages that follow the protocol synthesis rules. In order to make sure that generated protocol implementation is correct, we should use implementation tests which consist of generating test cases for the software modules, traditionally in the form of input/output sequences.

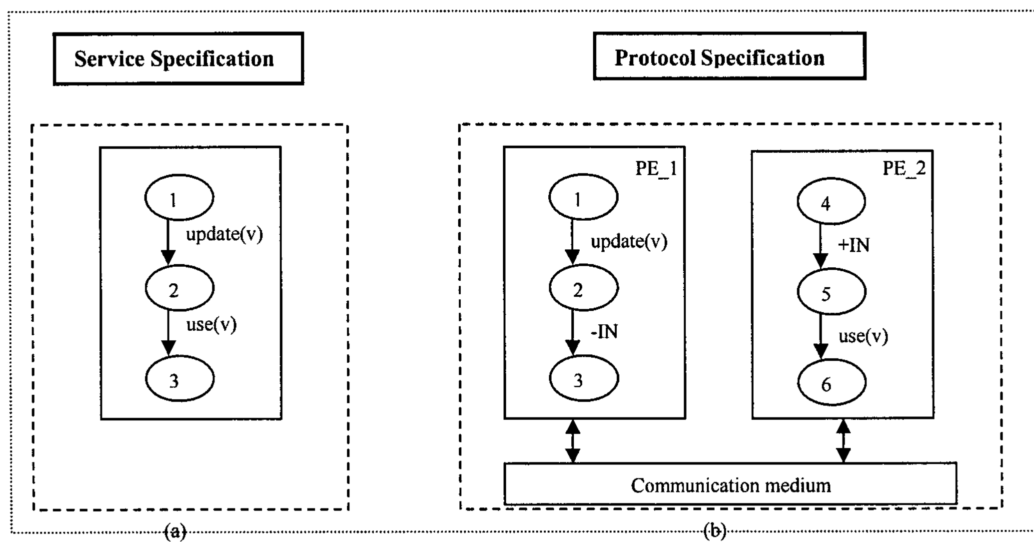


Figure 2.4: (a) service specification (b) protocol specification

2.2 Overview of Petri nets

Petri nets were developed in the early 1960s by C.A. Petri in his Ph.D dissertation as a mathematical tool for modeling distributed systems. Petri nets have been successfully used for concurrent and parallel systems modeling and analysis, communication protocols, performance evaluation and fault-tolerant systems, etc [35].

A Petri net is composed of four parts: a set of transitions T – drawn as bars, a set of places P – drawn as circles, a flow relation F which relates transitions and places and token which is graphically represented by a black dot. A place is a basic Petri net component that represents a condition. If a place is a member of the input function of a transition, it is a pre-condition for that transition, or event, to occur. If a place is a member of the output function of a transition, it is a post-condition of the event or transition firing. A transition is a basic Petri net component that represents an event and the flow relation is represented by directed arcs connecting places and transitions. Places can be marked by tokens, indicated graphically by marking the circle with a point (see Figure 2.5).

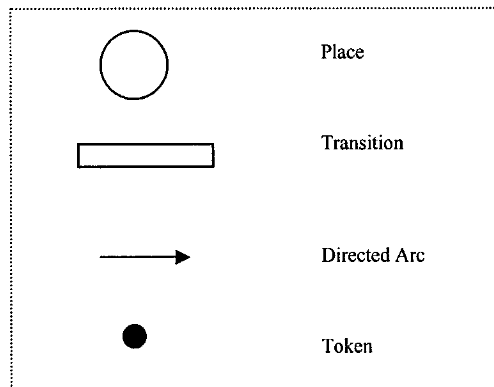


Figure 2.5: Basic symbol in Petri nets

DEFINITION 2.1: A Petri net, C, is a five-tuple, $C = (P,T,F,W,M_0)$, where

$P = \{ p_1, p_2, \ldots p_n\}$ is a finite set of places, $n \geq 0$.

$T = \{ t_1, t_2, \ldots t_m\}$ is a finite set of transitions, $m \geq 0$.

$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs.

$W: F \rightarrow \{1,2,3,\ldots\}$ is a weight function for arcs.

$M_0: P \rightarrow \{1,2,3,\ldots\}$ is the initial marking.

$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

The firing rules of Petri nets are:

- A transition t is enabled to fire if each input place p of t is marked with at least w(p,t) tokens, where w(p,t) is the weight of the arc from p to t.
- A firing of an enabled transition t removes w(p,t) tokens from each input place p of t, and adds w(t,p) tokens to each output place p of t.
- The marking of the other places that are neither input nor output of t remains unchanged.

We use a very simple Petri net as an example to show how these elements can be connected and fired. The weight of an arc indicates how many tokens are moved when the related transition fires. The weight is 1 by default (see Figure 2.6).

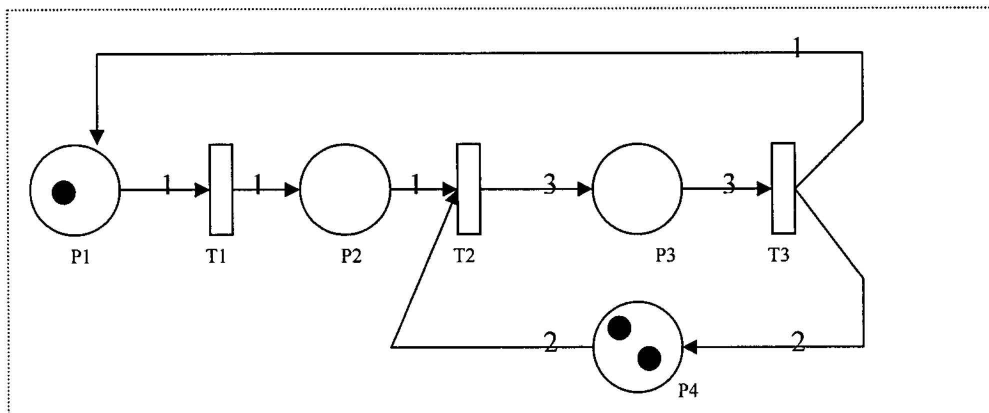


Figure 2.6: Petri net example

If a place is connected to a transition by a directed arc, the place is described as the input place to the transition; for instance P1 represents the input place to transition T1. Similarly, if there is a directed arc connecting a transition to a place, then the place is an

output place of that transition. Of course a single place can be connected to a single transition with more arcs. Conforming to these basic rules, multiple places and transitions can be connected to build very complex nets. So it can easily model a complex system's behaviour.

Let us look into the concept of marking: the marking describes the token distribution among all places of a Petri net. When one or more tokens reside in a place, the place is called marked, otherwise the place is unmarked. The number of tokens in a place represents the local state of the place so that the marking of the net represents the overall state of the system. Figure 2.6 captures the initial marking of the net, which represents the initial state of the system.

There are two parts in a Petri net model:

- The net structure: it represents the static view of the system.
- Marking: it represents the current overall state of the system.

The dynamic behaviour of the system is then modeled by the flow of tokens and the firing of transitions. The transition firing means that tokens are moved from the input places and new tokens are placed on the output places.

Let us look at a simple Petri net (see Figure 2.7) and see how a transition can be fired.

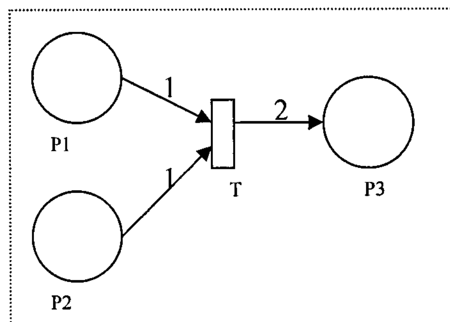


Figure 2.7: Simple Petri net structure

The firing of a transition follows these steps:

- The initial marking is shown in Figure 2.8.

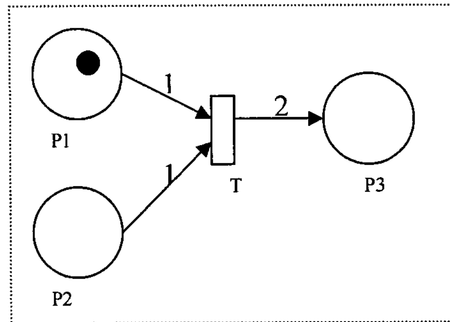


Figure 2.8: Initial state

- A transition is said to be enabled if the number of tokens in each of its input places is at least equal to arc weight going from the place to the transition. The enabled transition may fire at any time by removing one or more tokens from each input place according to the arc capacity (see Figure 2.9).

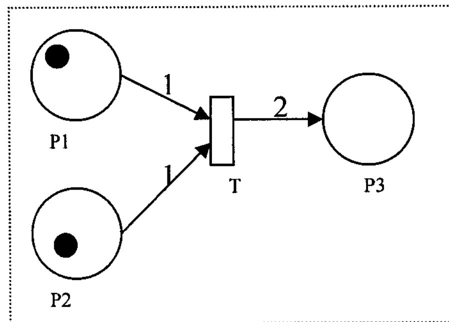


Figure 2.9: T is enabled

- Figure 2.10 shows the firing of T.

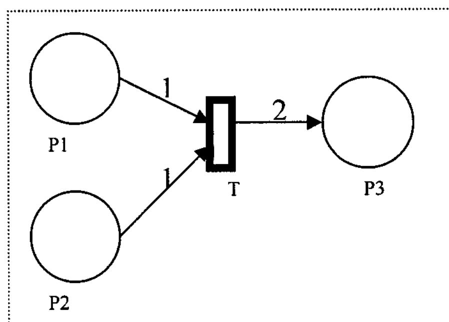


Figure 2.10: T firing (t → 0)

- When the transition is fired, new tokens will be added to the output places connected to the transition (Figure 2.11).

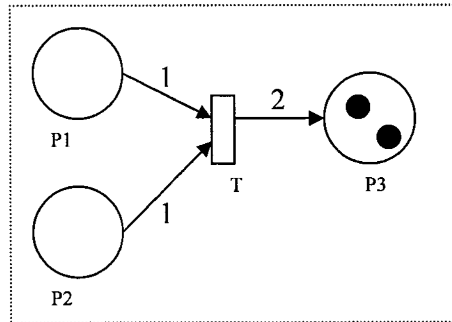


Figure 2.11: End of the firing

The above mechanism is usually called firing rule or "token game". While this token game governs the dynamic behaviour of Petri net models, the meaning of this process is determined by the net interpretation. When applied to different applications, the net elements may represent different things. In the protocol context, normally, a place represents the state of a resource or a message in transit. A token in a place would then represent an object of a class. Events and activities are then modeled by transitions. The firing of a transition is used to trigger an action or event, or to indicate the start or termination of an activity.

In the Petri net model, two events can be both enabled and may occur independently and concurrently. So unless there is a synchronized request by the underlying system that is being modeled, there is no need to synchronize events. When synchronization is needed, it is easy to model this too. Thus, Petri nets would seem ideal for modeling systems of distributed control with multiple processes executing concurrently in time.

The asynchronous nature is another major feature of Petri nets. There is no inherent measure of time or the flow of time in a Petri net.

Petri nets also have the ability to handle operation sequence, concurrency, conflict and mutual exclusion in system. These features make it a promising tool for describing and analyzing concurrent and real-time systems, such as a FMS (Finite State Machine).

Analysis of Petri nets can be based on enumerating all possible markings to form the reachability trees through methods and theories of discrete mathematics like matrix equations.

DEFINITION 2.2: (Behavioural properties of Petri nets)

1. Reachability: a Mn is reachable from M_0 if there exists a sequence of firings that transform M_0 into Mn.
2. Boundedness: a Petri net is bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0 for any reachable marking.
3. Liveness: a Petri net is live if, no matter what marking has been reached, it is possible to fire each transition with an appropriate firing sequence.
4. Reversibility: a Petri net is reversible if, for each marking M reachable from M_0, M_0 is reachable from M.
5. Persistence: a Petri net is persistent if, for any two enabled transitions, the firing of one of them will not disable the other; then, once a transition is enabled, it remains enabled until it is fired.
6. Conservation: a Petri net is persistence if the total number of tokens in the net is constant.

There are two main advantages of using Petri nets in system modeling and analysis:

With respect to modeling, Petri nets offer the following advantages:

- It is very simple and straightforward to model concurrency, conflict and mutual exclusion features using Petri nets.
- Users can use the formal graphical representation to model any complex system.

With respect to analysis, Petri nets have the following advantages:

- Petri nets have a well-developed mathematical foundation, so it can detect deadlock, etc.
- With the help of mathematical analysis of the model, we can do model performance evaluation easily.

2.3 Introduction to Coloured Petri nets.

Coloured Petri nets were developed in the late 1970s by K.Jensen (see [38]). The typical examples of application areas are communication protocols, distributed systems, automated production systems and work flow analysis. Each place contains a set of markers, called tokens in Coloured Petri nets. In contrast to ordinary Petri nets, each of these tokens carries a data value, which belongs to a given type. Each token often represents an object in the modeled system and each token has a value often referred to as 'color'.

Compared with Petri nets, Coloured Petri nets have these advantages:

- Coloured Petri nets allow the use of tokens that carry data values and can hence be distinguished from each other – in contrast to the tokens of ordinary Petri nets, which by convention are drawn as black, "uncoloured" dots.

- Coloured Petri nets combine the strengths of ordinary Petri nets with the strengths of a high level programming language. They provide the primitives for graphically defining process interactions while the programming language provides for the definition of data types and the manipulation of values of those data types.

As for all formal languages, the practical use of Coloured Petri nets is highly dependent on the tool support. So in this project, we use the tool: Design/CPN [41] to build our model and verify it.

CHAPTER 3 Protocol synthesis using Petri nets

3.1 Overview of protocol synthesis

There are two views of a distributed system. At a high level view, the system is viewed as a service provider that offers some specified services to users who access the service through many distributed Service Access Points (SAPs). At the low level view, the system consists of a number of cooperating Protocol Entities (PEs) that exchange messages. The PEs exchange these messages over a reliable communication medium according to a FIFO (first-in, first-out) discipline. Specifications at the former and latter levels are called a service specification (functions provided by the communication system) and a protocol specification (behaviour of the protocol entities). The relationship is shown in Figure 3.1.

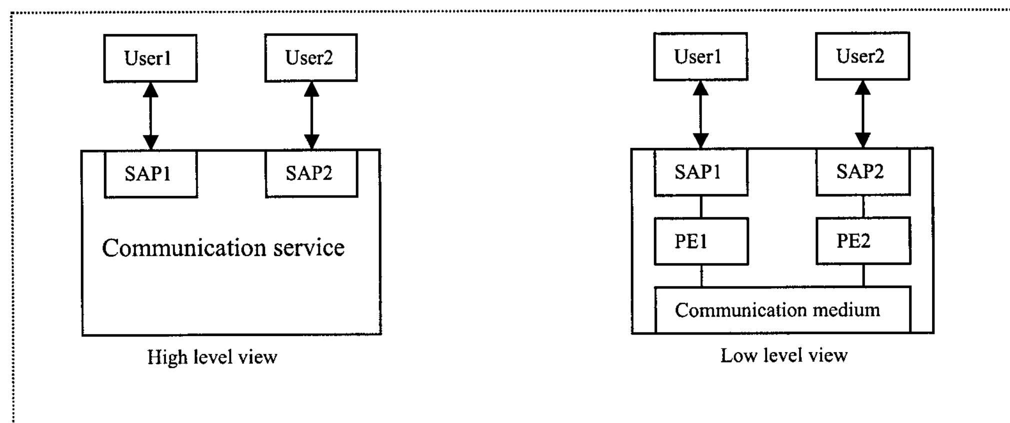


Figure 3.1: Views of service and protocol

There are three properties that must be guaranteed in protocol design:

- Safeness: it ensures that a protocol never enters an undesirable state, e.g. the protocol should be free from deadlocks.
- Liveness: it ensures that a protocol will enter a desirable state eventually; the protocol performs its intended functions with respect to the service specification.

- There are two other features of a protocol specification that should be satisfied: (a) timeliness with respect to the response time requirement of service specification and (b) fault-tolerance with respect to certain component failures or message loss.

Protocol synthesis focuses on generating an error-free protocol specification automatically from a higher-level service specification. The behaviour of the SAP provided by protocol entities should satisfy the requirement of service specification.

There are two approaches to the design of communication protocol: human design approach and protocol synthesis approach.

- In the human design approach, the protocol design begins from the preliminary version of protocol. The message exchanges among protocol entities are defined by hand, which sometimes leads to errors and deadlocks. Therefore, a verification process is needed to make this design become error-free. Another shortcoming of this approach is that the modification of an existing design is often very difficult.
- In the protocol synthesis approach, the protocol design is automatically constructed from the service specification (or completed) following a construction algorithm. The interactions among the protocol entities are generated without introducing any error and provide a correct protocol specification. So there is a big advantage – the correctness of the protocol should be a direct by-product of the protocol synthesis approach.

In protocol synthesis methods, a derivation algorithm automatically derives a protocol specification from a service specification. Often a resource allocation is also given, which specifies an allocation of SAP's and internal resources to the protocol entities. Many

papers propose protocol synthesis methods. In order to classify them, we can consider the following general features that characterize these methods:

- Starting point: The method may start from: (1) informal specification of requirements, (2) complete service specification, (3) complete specification of one protocol entity.
- Constraints of the communication model: (1) the interactions among the communicating entities and service users: the exchange of message is tightly coupled or loosely coupled to model the asynchronous behaviour of the service users, (2) the number of protocol entities that can be synthesized by the method:
- Interaction with the designer: The protocol design is automatically or the designer has to make choice during the design.
- Liveness and safety can be satisfied or not.
- Modelling formalisms: (1) Communication Finite State Machine (CFSM), (2) LOTOS (Language of Temporal Ordering Specifications), (3) Petri net, (4) Temporal logic.

The following papers are classified according to the starting point feature:

- Informal specification of requirements:

  [12] gives an interactive method to construct two communicating finite-state machines, which exchange message over two unidirectional FIFO channels when the function of the protocol has been given. The method is based on the production of a global state transition graph of a protocol to be synthesized and then produces the protocol.

[14] starts from the designer's specification of sending events in protocol entities and synthesizes based on some design rules.

- Complete service specification:

  [4] gives an algorithm to automatically synthesize a correct protocol entity specification from a service specification in a Petri net model with registers. In order to simply the algorithm, the control flow of a service specification should be a free-choice net.

  [5] gives a method for deriving protocol specifications from given service specifications with time constraints. This algorithm can derive a correct protocol specification from a service specification described in time Petri nets model and can treat the class of service specifications with both parallel synchronization and data values.

  [6] presents a new method for synthesizing protocol specification from service specification in an interpreted Petri net model. The model can handle the control flow, data flow, distributed choice and data coherency constraints simultaneously. This method is based on stepwise refinement rules (basic rules and contextual rules). The basic idea of this synthesis strategy is to refine either a place or transition object of the service specification by constructing sub-Petri-nets.

  [8] presents an algorithm that fully automates the derivation of protocol specifications from service specifications. The service specification is described by a set of LOTOS operators and is represented by a syntactic tree. And it can be automatically transformed into the specification of the protocol entities.

[9] starts from a FSM service specification and synthesizes two-entity FSM protocol specification. The method was extended to synthesize error-recoverable protocol specifications.

[10] starts from a service specification modeled by FSM and synthesizes protocol entities. The method starts from assigning the distributed service to each SAP. Then one applies transition synthesis rules to generate message transitions. The result satisfies liveness and safety.

[11] starts from a LOTOS-like description of the service specification which is then transformed to a relational table representation of the service.

[24] starts from a propositional temporal logic formula specifying in the sequence of communications. It uses a tableau-like decision procedure for propositional temporal logic to synthesize the synchronization part of the communicating processes.

[25] begins from a branching time temporal logic specification of the communication system and synthesizes the synchronization part of shared memory programs achieving the required service.

- Complete specification of one protocol entity:

  [7] is the first method that can generate protocol entity automatically. It starts from manually constructing finite state machines representation of a protocol entity and solves the problem – we have n-1 communication finite state machines and a service specification, how can we get the n-th communication finite state machine. The n communication finite state machines obtained satisfy the given service specification.

[13] gives some production rules for the interactive synthesis of protocol entities. The protocol entity is modeled by finite state machine.

[15] The method starts from an uncompleted FSM specification of a protocol entity, and then synthesizes two complete protocol entities.

[16] gives a method to create a Petri net of the peer entity of a two-way protocol automatically. First the approach designs one party, then verifies the correctness of the model. If the model is correct, then we can apply a set of well designed transformation rules to generate the corresponding peer entity. If the given entity model follows certain desirable properties, the generated peer entity is guaranteed to be correct.

3.2 Petri nets based protocol derivation

3.2.1 Service decomposition for distributed system design

Three papers discuss the service decomposition for distributed system design:

[4] gives a protocol synthesis method based on a Petri net model with registers. The control flow of the model is specified by a free-choice Petri net. The free-choice Petri nets are Petri nets such that every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition. So the synthesis method does not consider the distributed choice problem. The control flow is described as a Petri net and the model has a finite number of registers that represent resources in the distributed system. A correct protocol entity specification for each node can be derived automatically from service specification.

For each transition t in the given service specification, we can construct a sub-Petri-net that simulates the transition t. A protocol entity is constructed by replacing each transition t in service specification with the corresponding sub-Petri-net, see Figure 3.2.

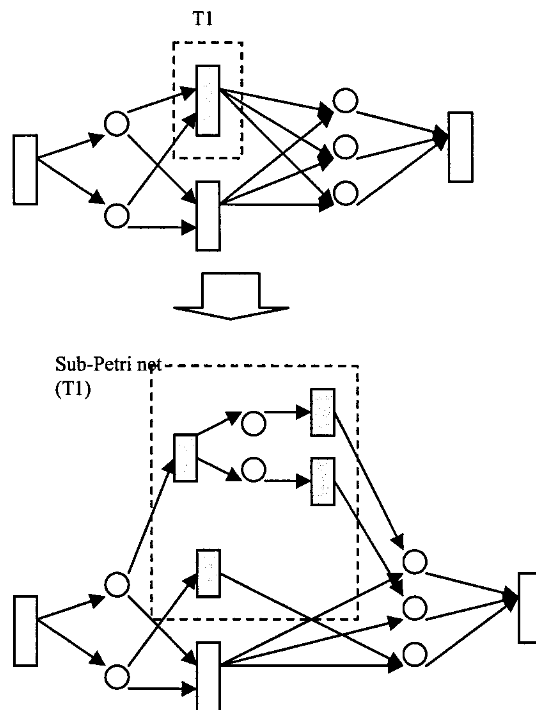


Figure 3.2: Replacement of transition T1

However, the expressive power of the service specification model in [4] is limited by using restricted Petri nets. The free-choice Petri nets forbid the use of distributed selective structures. And the access to a register cannot be concurrent.

[6] addresses these issues. The paper gives an approach that is based on stepwise refinement rules. The strategy of this approach is to refine either a place or a transition object of the service specification by using a set of refinement rules to construct n cooperating sub-Petri-nets. It addresses control flow synthesis, data flow synthesis,

distributed choice, etc. However, since [4] and [6] mainly focus on value exchanges between entities, only simple control flows are allowed and the combination of choice and synchronization involving parameters is not allowed.

[40] extends this work. It gives a new algorithm for the derivation of a protocol specification in Pr/T-nets (Predicate Transition Nets) from a given service specification. The new algorithm can derive a protocol specification in Pr/T-nets from a service specification in Pr/T-nets and an allocation of the places of the service specification to N entities. The method can preserve the properties of the Pr/T service specification and also includes a mutual exclusion algorithm. This synthesis method can handle the distributed choice in Petri nets and generate correct protocol specification from service specifications.

3.2.2 Protocol derivation example

In [4], the registers represent resources in distributed system. At each transition, one I/O event is executed and the values of the next registers are calculated from the values of the current registers and input data. We can have a guard to control the firing of a transition. Let us look at the service specification in Figure 3.3:

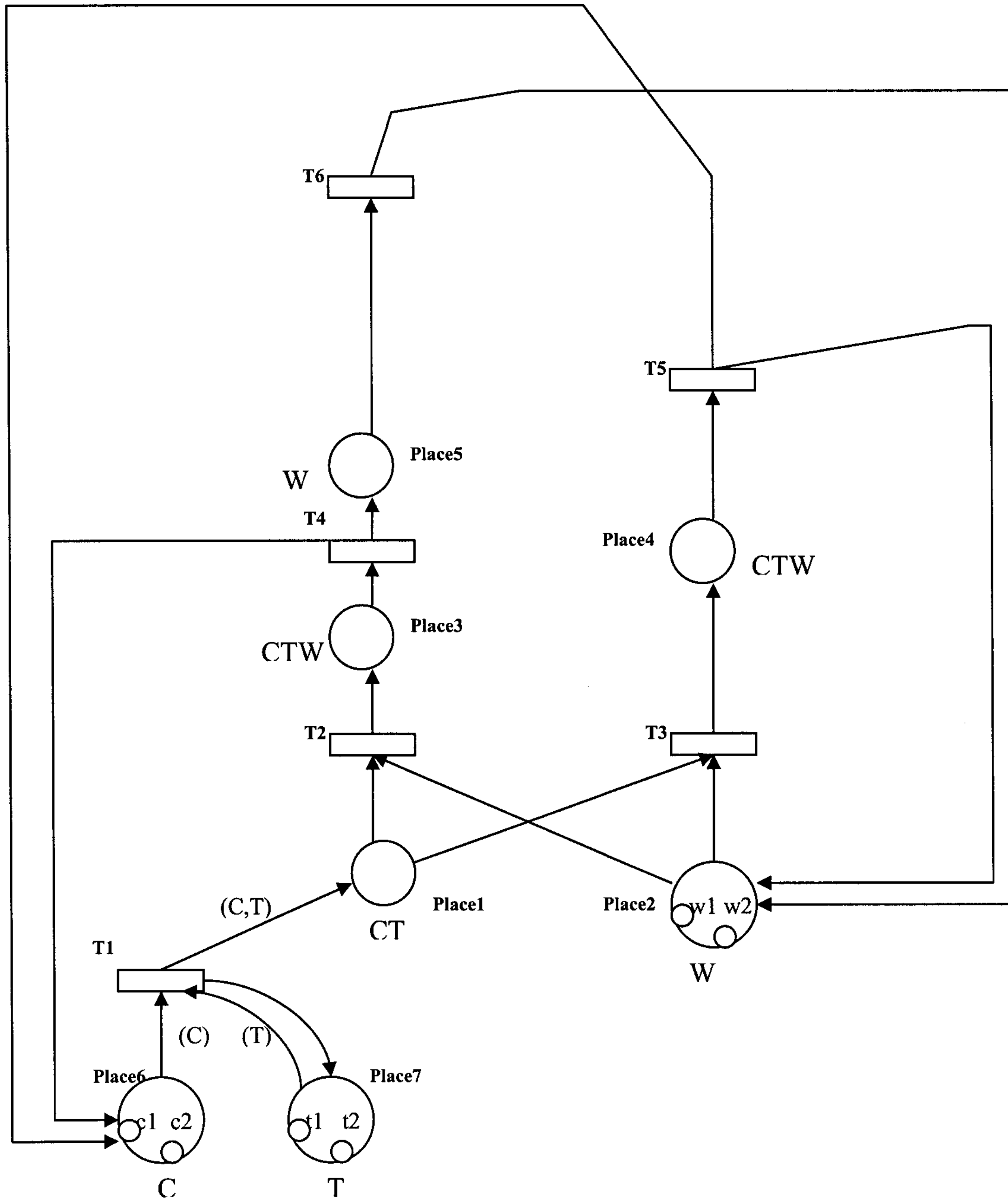


Figure 3.3: Service specification example

Figure 3.3 shows the specification of a task broker system. In this system, there are three types of tokens – Client, Task and Worker and each type of token has two instances, such as c1 and c2 in Figure 3.3. Clients can search for Workers to perform some Tasks. The

pair of a Client and the requested Task is represented as a token of CT type (see Figure 3.3) and they wait for a suitable Worker for doing the Task. Workers can select a Task from the requested Tasks and perform it. If the Task is a collaborative Task, the Client and the Worker work together and afterwards the Client will pay the Worker for the task. If the Task is not a collaborative Task, the Client pays for it before the task is done. The Worker will automatically receive the reward of the Task after finishing the Task. In this case the client can request the next Task without waiting for the last Task to be finished.

When the condition of each transition is satisfied (each transition has enough input tokens), the transition can be fired. For example in Figure 3.3, the T1 transition can be fired, but T2 and T3 transitions cannot be fired because there is no token in Place1,

The tokens in the place Tasks will be returned back when transition T1 fires. Each token in the place Clients will request a token from the place Tasks. After that, the token will be sent to Place1. When the tokens are available, the transition T2 or T3 will be fired, depending on the type of the Task. If the tokens in the place Place2 are not available, T2 and T3 cannot be fired and tokens in Place1 will have to wait until they are available.

In order to avoid distributed choice (that is, a transition with input places allocated to different sites), new places and transitions are created in the derived service specification. The derivation of protocol specification from service specification was done by Osaka University, Japan. The derived service specification that includes three sub-Petri-nets is shown in Figure 3.4.

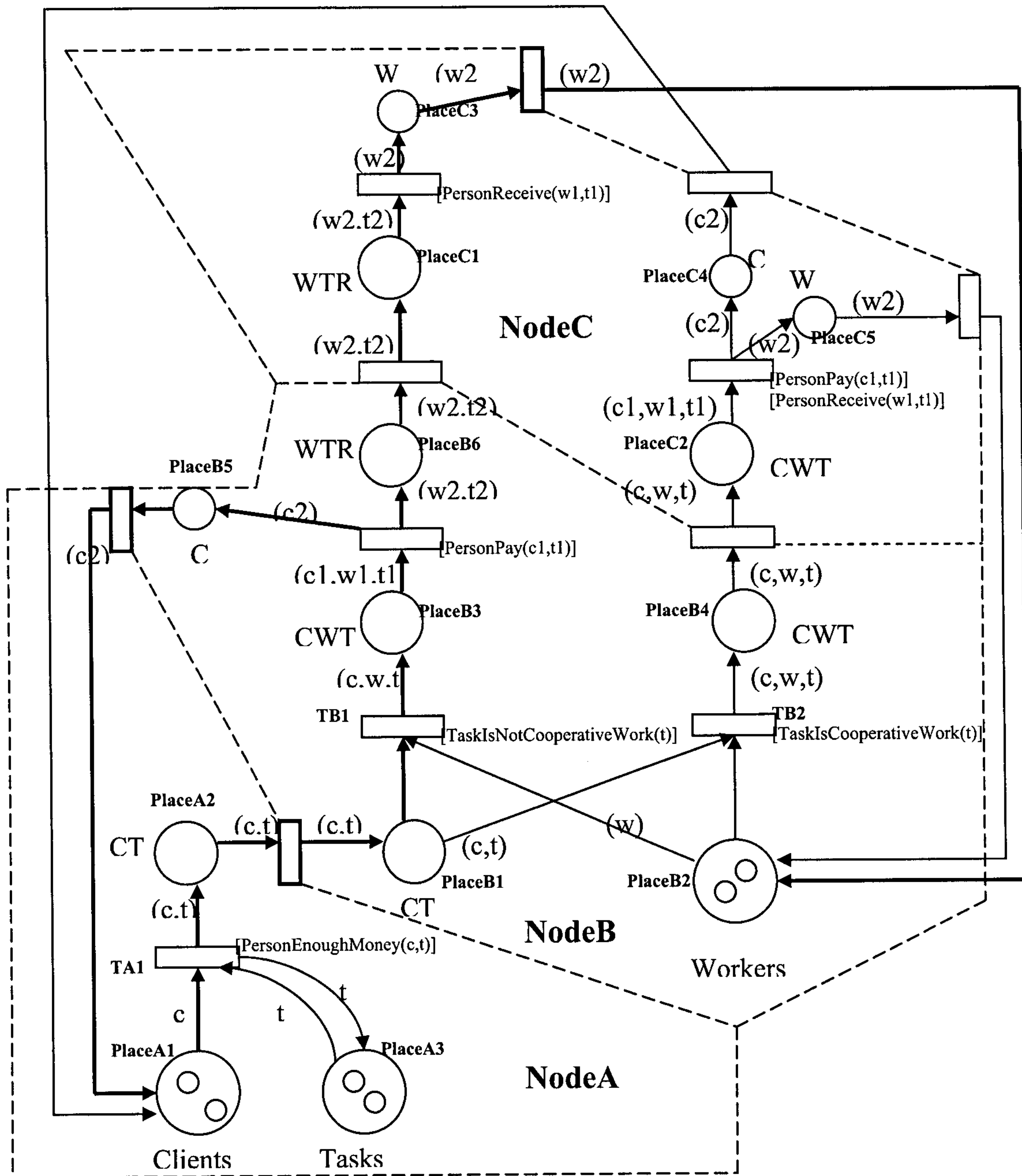


Figure 3.4: Derived service specification

Let us look at the control flow of the darker lines in Figure 3.4:

If the two places (Clients,Tasks) have at least two different types of tokens and the function-PersonEnoughMoney() returns true, the transition-TA1 will be fired and new

tokens will be moved to PlaceA2 and then moved to PlaceB1. The relationship of PlaceB1 and PlaceB2 is synchronization, which means only PlaceB2 has a token available and token type is satisfied then TB1 will fire otherwise token in PlaceB1 will wait.

The relationship of PlaceB5 and PlaceB6 is concurrency: after the action-PersonPay(), one client token will be moved to PlaceB5 and two tokens (task,worker) will be moved to PlaceB6. The token in PlaceB5 will be returned back to Clients. After the action-PersonReceive, worker token will be returned back to PlaceB2.

For our protocol specification example, there are three different processing regions: NodeA, NodeB and NodeC. Every processing region has a simulation engine that can process all the requests from other regions and generate corresponding responses.

The communication interfaces are showed with the darker rectangle.

The CPN ML [53] language is used to model our example. It is a typed functional language that uses types, functions, operations, variables and expressions. This language is the extension of a well-known functional programming language, called Standard ML (SML). Standard ML is a type-safe programming language that embodies many innovative ideas in programming language design.

CPN ML has compound color sets. For example, the following is the definition of a product color set:

Color A = product $name_1$ * $name_2$ * …* $name_n$.

The compound color sets in Figure 3.4 are defined as follows:

color CT = product Client * Task;

color CWT = product Client * Worker * Task;

color WTR = product Worker * Task;

color C = Client;

color W = Worker;

The token distribution among different Petri nets needs communication interface. The communication interface is used to communicate with other Petri nets.

The example Petri nets in Figure 3.4 include three sub-Petri-nets. We distribute the three sub-Petri-nets in different nodes to generate derived protocol specification.

We applied the rules of common region (see Section 4.1) to Figure 3.4 and divided it into three subsets of Petri nets regions: NodeA, NodeB and NodeC. In order to avoid the distributed choice, the transition of TB1 and TB2 were set in the same region.

The derived protocol specifications including communication interfaces are shown in Figure 3.5, Figure 3.6, and Figure 3.7.

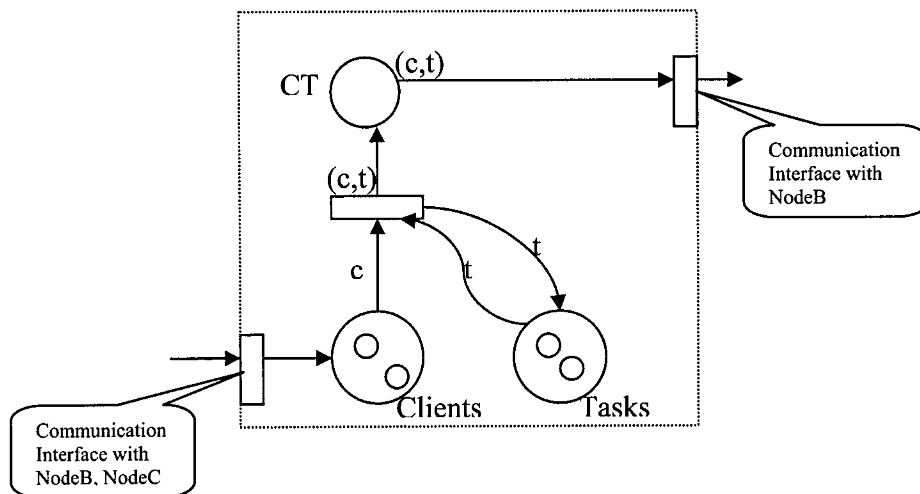


Figure 3.5: NodeA

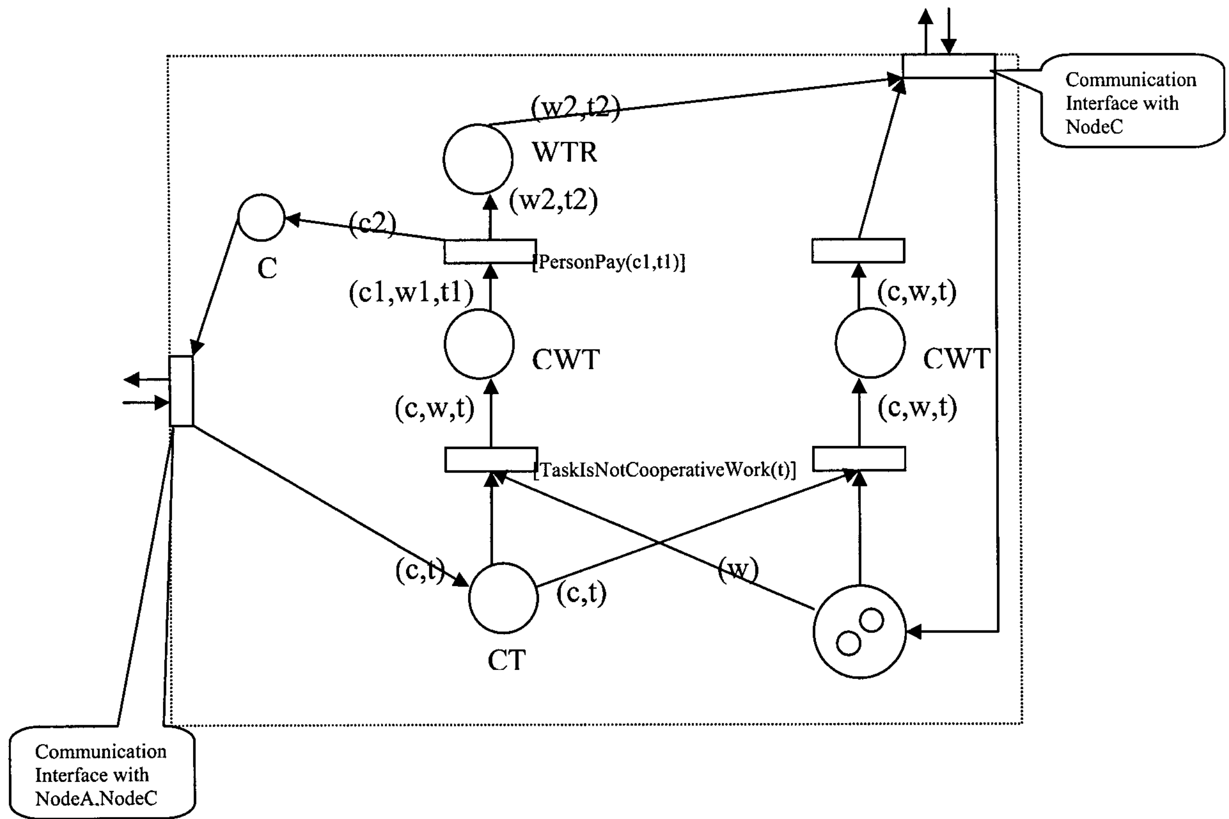


Figure 3.6: NodeB

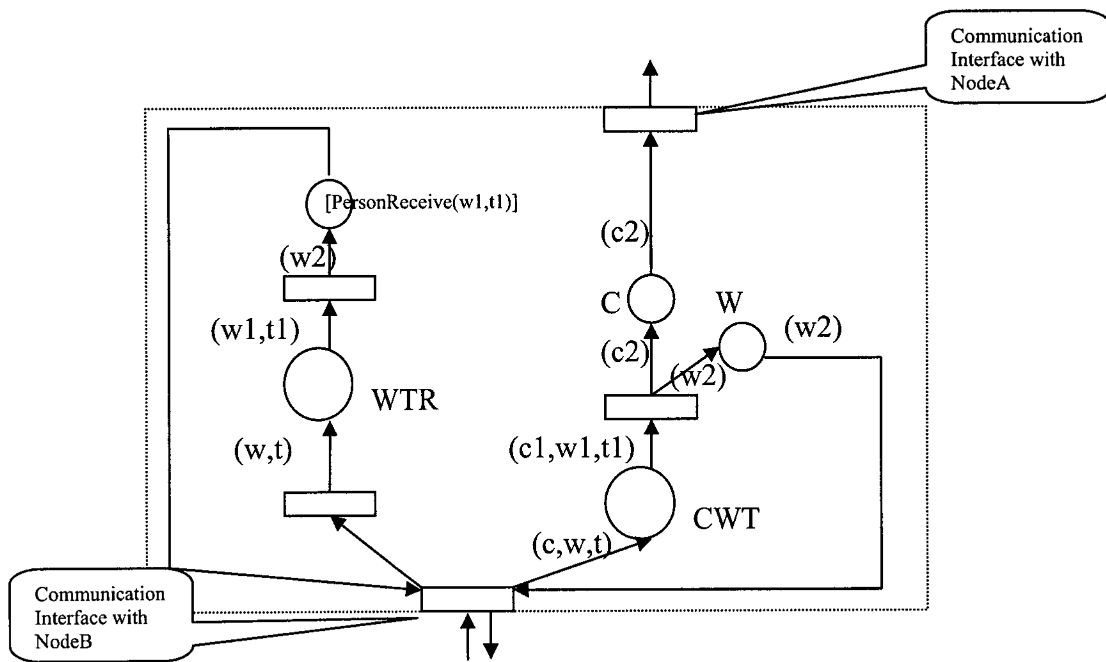


Figure 3.7: NodeC

CHAPTER 4 Distributed Petri net implementation methodologies

4.1 General methodologies of implementation

The Petri nets describe algorithms both in graphic and mathematical ways. The visually appealing graphical representation makes Petri nets easy to understand and to be manipulated by the developers. Petri nets support concurrent and asynchronous features that are inherent to distributed systems. There are lots of models and verification technologies and computer-aided tools based on Petri nets (see [52]).

Distributed Petri nets can be viewed as a distributed network of logic processes, where usually each logic process realizes a part of the distributed Petri nets. The communication among processes is via messages.

For distributed Petri net implementation, the whole Petri net is divided into several sub-Petri-nets, each sub-Petri-net resides on a different machine running one process. The process will reside there for the whole execution time. Each process performs automatically the behaviour of the corresponding sub-Petri-net and communicates with the other processes by sending and receiving messages through a reliable transport protocol.

For the most general case of distributed Petri nets control flow, we can decompose a sequential task into a set of logic processes assigned to individual processing regions that work automatically and communicate with each other for synchronization. A RPC-based message transport system implements the directed, reliable, communication channels between the processing regions.

Each sub-Petri-net may include concurrent activities. They may be implemented by using multiple threads in each logic process. We propose a distributed Petri net implementation framework that realizes a sub-Petri- net in terms of a communication interface and a logic process that reside on a given machine.

The decomposition of Petri nets strongly affects the execution performance. If the regions are small, the communication demands will cost lots of system resource and realize system performance. Meanwhile, a large region can reduce the network traffic significantly, but require more processing power.

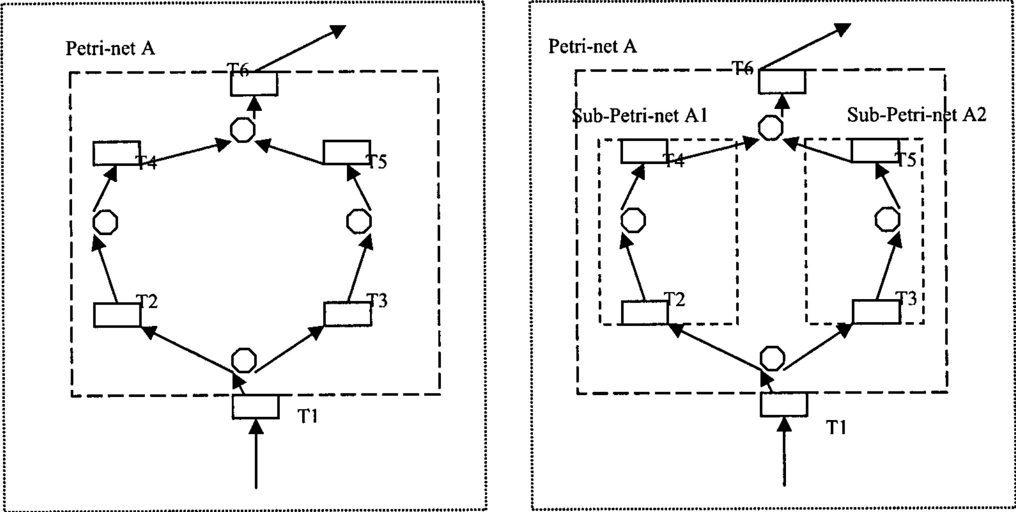


Figure 4.1: Decomposed Petri nets

In the Figure 4.1, the Petri net A has six transitions: T1, T2, T3, T4, T5, and T6. The T2 and T3 transitions are fired randomly. If we decompose the Petri net A into two sub-Petri-nets: sub-Petri-net A1, sub-Petri-net A2 as shown in the Figure 4.1, four communications interfaces (T2,T3,T4,T5) are needed for the sub-Petri-nets to receive and send tokens. This will slow the execution performance. Therefore T2, T3, T4, T5 should reside in a common region (Petri-net A) as shown in Figure 4.1.

We call common region a set of transitions of a Petri net for which the decision of firing can be done solely based on local information. So any set of conflicting transitions (see Section 4.2) should reside in a common region.

We can follow the rules below to set up minimum regions:

- Conflicting transitions should reside in a single sub-Petri-net in order to avoid distributed choice.
- In order to save network traffic, we should put the transitions and places with high message exchange into a single sub-Petri-net (as Figure 4.1).

In order to ensure the orderly exchange of tokens between the different regions of a given Petri net and to allow for a distributed implementation where the different regions are implemented as sub-Petri-nets on different computers, we introduce the notion of a communication interface. A communication interface is an additional transition that represents the transmission of token between two sub-Petri-nets and used to preserve the behavioural semantics of the whole Petri nets. It is the channel between two sub-Petri-nets – corresponding to all the arcs from transitions in one sub-Petri-net to another sub-Petri-net. All the tokens sent from one sub-Petri-net to another specific sub-Petri-net will go through the same interface. The movement of received tokens follows the directed arc that connects the place and the transition in the respective sub-Petri-net. In our distributed system implementation, communication between two sub-Petri-nets is implemented by remote procedure call (see Section 4.2.2) and all the tokens pass the communication interface as arguments and return values of remote procedure calls.

There is only one logic process that runs on each subset of Petri nets region. The logical process creates and maintains the objects of the places, also it initializes the control flow

of the subset of the Petri net (see Section 4.2). The logic process can execute the firing behaviours of the transitions in the region following the Petri net semantics by using multiple threads: When it gets input tokens, it will create a new thread to move the input tokens to the corresponding place which is linked by the arc (from the place to the transition). According to the control flow of the Petri nets, we may have different mappings from place to transition:

- The mapping of the place and the transition is 1:1

  Because the place has the tokens, the transition is ready to be fired. Since we use function call to model transition in our model, a function call will be executed.

- The mapping of the place and the transition is 1:N

  In this situation, the tokens in the place can be passed to one of the n different transitions randomly.

- The mapping of the place and the transition is N:1

  For this case, the transition can only be fired until all the n input places have enough tokens. The place that has the input tokens will continuously check the other n-1 places that whether they have enough tokens or not. If yes, the transition can be fired, otherwise the new thread will keep waiting.

Similarly, we can have the different mappings from transition to place:

- The mapping of the transition to the place is 1:1

  The tokens will be passed to the place.

- The mapping of the transition to the place is 1:N

  The thread created by the logic process will create other n threads to move the tokens to the n places simultaneously.

- The mapping of the transition to the place is N:1

  All the n transitions can pass the tokens to the place.

After the firing of the transition, the token is passed to another place which is linked by the arc (from the transition to the place). The new thread sends the tokens to another logic process using communication interface and is deleted afterwards. The detailed discussion of how the token is moved in the sub-Petri-net region can be found in the next section. The logic process has two queues: input queue and output queue. It will put incoming messages into the input queue and put generated messages into the output queue.

4.2 Local Petri nets control flow analysis

4.2.1 The relationships between places and transitions in Petri nets

As we discussed in Chapter 2, Petri nets are useful for describing and studying systems that are characterized as concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. In addition, tokens are used in these nets to simulate the data flow of real systems. As a mathematical tool, it can have state equations, and other mathematical models modeling the behaviours of systems.

The Petri nets execution is viewed as a sequence of discrete events. The order of occurrence of the events follows the basic structure of the net. This often leads to nondeterminism in Petri net execution. If more than one transition is enabled at any given time, then any of the enabled transitions may be the next to fire. From the point of view of the execution model, the choice about which transition will be fired is nondeterministic. This feature of Petri nets reflects the fact that in real life situations where several things are happening concurrently, the order of occurrence of events will be any of a set of sequences of events may occur during the execution period.

Because of the existence of concurrency, communication, synchronization and nondeterminism in Petri nets execution, the implementation of Petri nets control flow becomes complicated indeed.

During the implementation stages, we need to concentrate on the major operations and their sequential, concurrent, or conflicting relationships.

The basic relationships are:

- Sequential: If one operation follows the other, then the places and transitions representing them should form a cascade or sequential relation in Petri nets. It is showed in Figure 4.2.

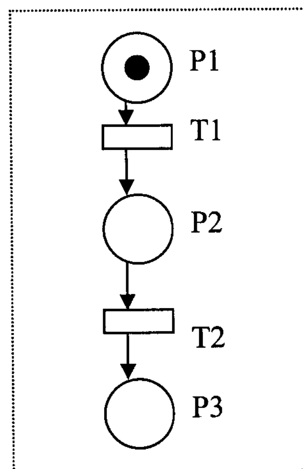


Figure 4.2: Sequential relation

- Concurrent: If two or more operations are initiated by an event, they form a parallel structure starting with same transition; two or more places are the outputs of a same transition. The pipeline concurrent operations can be represented with a sequentially connected series of places/transitions where multiple places can be marked simultaneously or multiple transitions are enabled at certain markings. This is shown in Figure 4.3.

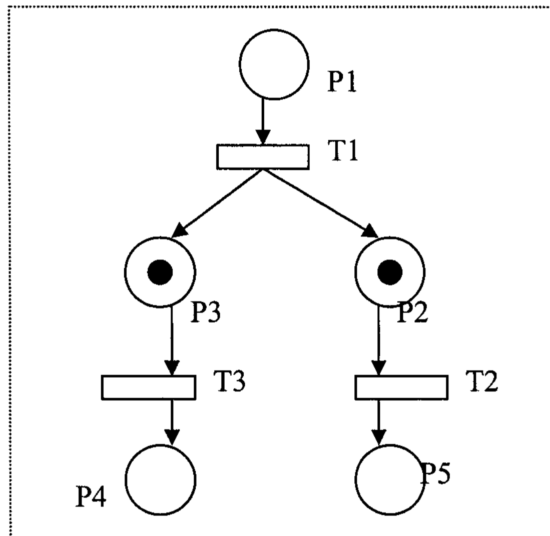


Figure 4.3: Concurrency relation

- Mutual exclusion: When two or more processes want to access shared data, we need mutual exclusion to guarantee the data persistence: Mutual exclusion is a technique of defining entry and exit code so that only one process can access a shared data object at a time. It first waits until no other process is executing the shared data. Then it will lock the access to the share data, preventing any other process from it. It will modify the shared data and unlock it to allow other processes to access it. This is very important for the shared objects.
- Conflicting: If either of two or more operations can follow an operation, then two or more transitions form the outputs from the same place.

  It can be seen at Figure 4.4.

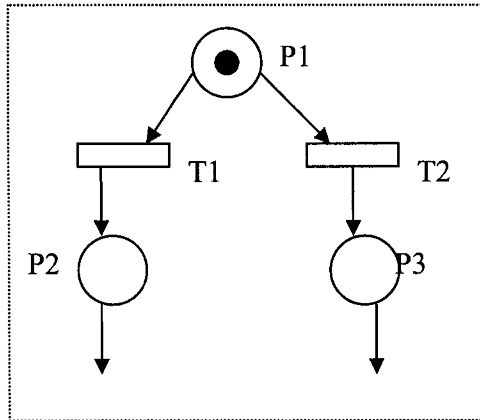


Figure 4.4: Conflict relation

- Synchronization: If a transition has two input places and only one place has a token, the transition cannot fire unless the other place gets a token. The resulting synchronization of activities can be captured by the transitions shown in Figure 4.5. Here, T1 will be enabled only when a token arrives into the place on the left. The arrival of a token into this place could be the result of possibly complex sequence of operations elsewhere in the rest of the Petri net model.

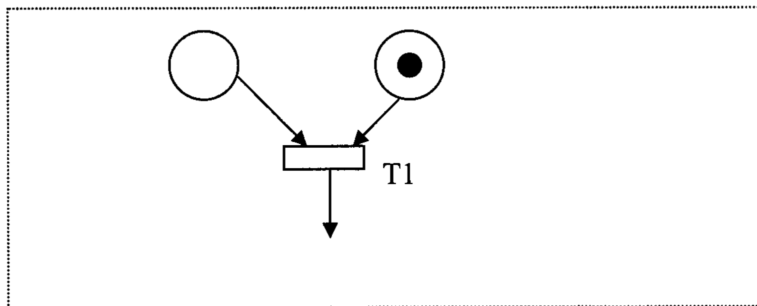


Figure 4.5: Synchronization relation

- Confusion: It is the combination of conflict and concurrency, as shown in Figure 4.6. We can see that T1, T2 and T3 are enabled at the same time. If T1 fires, T2 will no longer be enabled. It can be seen in Figure 4.6.

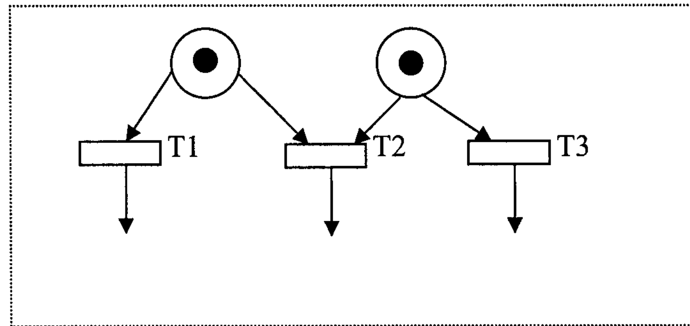


Figure 4.6: Confusion relation

Some solutions for the implementation of sequence, concurrence, synchronization and conflict relations can be found in next part.

4.2.2 Object-oriented modelling of Petri nets

There are several different programming techniques in software engineering. Procedure-oriented programming that was very popular software development paradigm in 1970s focuses on subprograms and subprogram libraries. Data-oriented programming focuses on abstract data types. In this paradigm, computation on a data object is specified by calling subprograms associated with the data object. Those couples lead to object-oriented programming which was first supported by the programming language SIMULA 67, and which was further developed in Smalltalk 80.

A language that is object-oriented comes with the three features:

- Abstract data types
- Inheritance
- Dynamic binding

Abstract data types, with encapsulation and access controls, are the units to be reused in object-oriented programming. But there is the problem of reusing abstract data types: the feature and capability of the existing abstract data types are not right for new use. So inheritance is the solution to the problem. The new abstract data type can inherit the data and functionality of some existing type, and is also allowed to modify functionality and add new data.

The abstract data types are usually called classes and a class instance is called an object. Dynamic binding is supported by the dynamic binding of messages to method definitions. The parent class can define a method that can be overridden by its subclass. When such method is called through a polymorphic variable, that call is dynamically bound to the method in the proper class.

In our distributed Petri-nets implementation, we use different classes for modelling different places and functions for transitions.

The class definition of a place follows the service specification example (Chapter 3 Figure 3.3). For example, the place PlaceA1 is represented by a class.

```
public class PlaceA1 {
public PlaceA1 () { ... } // construction method
public void addRef( PlaceA2 placea2,PlaceA3 placea3)    { .. } // add obj reference
public synchronized void addTokenA1(Client token)    { ....} // add token
public  synchronized void delTokenA1() { ..} // delete token
.. }
```

4.2.3 Concurrency in Petri nets

Concurrency can be divided into instructional level, statement level, unit level and program level [57]. Concurrent execution of program can occur either physically on separate processor or logically in a time-sliced single-processor computer system.

A process is a running program. The operating system allocates all resources that are needed for the process. A thread is a dispatching unit within a process. That means that a process can have a number of threads running within the scope of that particular process. Main difference between threads and processes can be defined as follows:

While a thread has access to the memory address space and resources of its process, a process cannot access variables allocated for other processes.

This basic property leads us to a number of conclusions:

- Communication between threads created within a single process is simple because these threads share all the variables. Thus, a value produced by one thread is immediately available for all the other threads.
- Threads take less time to start, stop or swap than processes

As we discussed, Petri nets are a powerful tool to model distributed, concurrent systems. The control flow in Petri nets can be: sequence, concurrence, synchronization or conflict. Our solutions for these relationships are discussed below:

- Sequence: we can use observer pattern or object reference to implement the sequence relationship.

  The idea of the observer pattern is that when an object's status changed, other registered objects can get a notice automatically. A Java solution for the pattern is the Observable class and the Observer interface. In the example, PlaceA1 is an Observable and PlaceA2 is an Observer. When the state of PlaceA1 is changed, PlaceA2 will automatically get the notification.

  The idea of object reference is that an object may have the references of other objects. We create an instance for each place class during program initialization

and each class has the function: addObject(). For instance, PlaceC1 and PlaceC3 have a sequential relationship, during the program initialization, the instance of PlaceC1 will have an instance of PlaceC3 using the function: addObject(). If a token will move from PlaceC1 to PlaceC3, we can call PlaceC3 instance's function – addToken() to achieve it.

- Concurrence: There are two concurrent executions in our example: local execution concurrency and distributed execution concurrency. We can find the local concurrency in the protocol specification example (Figure 3.3 in Chapter 3). The idea of distributed execution concurrency is that each processing node cannot be blocked at run time; the processing node can send and receive messages at the same time. In our implementation, we use Java multiple threads to implement these concurrent executions.

  Each processing node has a main process and listens to the messages coming to the listening port. When a new request comes, the main thread will create a new thread to handle the request and then continues to listen on the port. So at any time, the processing node may have multiple threads to handle different requests simultaneously and will not be blocked. If many requests come at the same time, much system resources will be consumed.

- Synchronization: The relationship of TB1, PlaceB1 and PlaceB2 in the example is synchronization. If there is no token in PlaceB2, TB1 will wait until it becomes available. In our implementation, when a new request comes to NodeB, the logic process running in the NodeB will create a new thread to move tokens to PlaceB1. If no token is available in PlaceB2, the thread will be blocked and wait. The

thread will continue checking PlaceB2 to see whether any token becomes available. If PlaceB1 and PlaceB2 both have tokens, TB1 will fire.

- Conflict: The relationship of TB1, TB2 and PlaceB2 in our example is conflict. When a token comes to PlaceB2 (PlaceB1 already has tokens), both TB1 and TB2 can be fired. The token of PlaceB2 can go to TB1 or TB2, a conflict exists.

In order to implement the synchronization relationship of Petri nets, we need mutual exclusion. Let us look at the idea of mutual exclusion: Two processes are mutually exclusive if they cannot be performed at the same time due to constraints on the usage of shared resources. Mutual exclusion is usually implemented by defining entry and exit code so that only one process can access a shared data object at a time. It first waits until no other process is executing the shared data. Then it will lock the access to the share data and prevent any other process from it. It will modify the shared data and finally unlock it to allow other processes to access it.

Because our implementation is highly concurrent, it is very important to make the shared data safe among threads. In our implementation, we use the Java keyword: synchronized to achieve mutual atomic execution. The semantics of synchronized in Java guarantees that during the execution of the method no other thread has access to the variables of the object instance. The synchronized keyword is added to each function that accesses the shared data.

4.2.4 Distributed implementation of Petri nets

The communications between programs are expressed as messages or remote procedure calls between objects.

- Message passing is a widely used standard paradigm for the communication between processes in a parallel or distributed system. The basic ideas have been around since the late 1960s. It allows a program to communicate with another program using messages. The disadvantage of message passing is that programmer has to explicitly code all synchronization among messages.
- Remote Procedure Call (RPC) is the basis of distributed computing, the way for one program to make a procedure call to another program, passing arguments and receiving return values. Both programs reside in different computers.

  The information passed between these computers must follow some agreed protocol. A procedure call is realized by the exchange of the messages:

  - RPC request message: it contains identification of the process, name of the method and name of input parameters.
  - RPC response message: it contains result parameters of the call or the indication of exceptions.

RPC uses existing low-level transport protocols (TCP/IP or UDP) for carrying the message data. It enables users to work with remote procedures as if they were local. In RPC, each server supplies a program that is a set of remote service procedures. The combination of a host address, program number, and procedure number specifies one remote service procedure. In the RPC model, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, and sends a reply back to the client. The procedure call then returns to the client.

The RPC interface is generally used to communicate between processes on different workstations in a network. However, RPC works just as well for communication between different processes on the same workstation.

To develop an RPC application the following steps are needed:

- Specify the protocol for client-server communication
- Develop the client program
- Develop the server program

Three ways to pass parameters are supported in RPC:

- Call by value: A value parameter is copied onto the stack. The client stub copies the value from the client and packages into a network message.
- Call by reference: A reference parameter is a pointer to a variable in memory (i.e. it is an address), rather than the value of the variable. But a pointer is meaningful only within the address space of the process in which it is being used. For instance, we cannot pass a pointer that contains the address of the buffer on the client to the server and expect it to work on the server. So this approach works well if the client and server have shared memory. The implementation is very difficult due to the different address spaces of the client and server. But we can use proxy to access remote object reference in Java RMI. A proxy in Java RMI is the local representative of the remote object being accessed by a client. Proxy implements the object's remote interface and typically forwards most of the calls on the interface to the remote object.
- Call by copy/restore: A variable is copied to the stack by the caller, as in call by value, and then copied back after the call, overwriting the caller's original value.

RPC transparency includes parameter passing, data representation and binding. RPC achieves the transparency by using stubs. A stub is a local representation of the remote object instance. It looks like a local procedure but actually takes care of the sending and receiving of the variables to the remote procedure. It hides message-passing I/O from the programmer.

The following RPC protocols are quit popular and are often used for the design of distributed software:

- CORBA (Common Object Request Broker Architecture) [54]. It was designed to realize the RPC for object-oriented systems where different programming languages are used in different computers.
- Java RMI [29].
- SOAP (Simple Object Access Protocol) [49].

The advantage of RPC is that it is request/response oriented and easier to use than message passing (I/O oriented). So we chose the RPC approach for the distributed implementation of Petri nets. The more detailed RPC interface definitions can be found in Chapter 6.

4.2.3 Layer pattern

The layers pattern is used in our Petri net implementation: The idea of the layer pattern is to structure the system into an appropriate number of layers and place them on top of each other. A well-known example is the OSI 7-layer model of communication protocols. The advantage of this pattern is its enhanced maintainability and extensibility. The layer pattern has the characteristic that all dependencies flow downward. This will avoid cyclic dependencies.

In our project, we have several layers in our design. The top layer is the presentation layer which has some simple Java GUIs to display the running results of the execution, the second layer is the business layer which has the Petri nets control flow model and other objects, the third layer is the integration layer which connects to database with JDBC and saves each transaction information, the last layer is resource layer which has MySQL database. Each layer communicates with the most-lower layer by using function calls. This structure is show in Table 4.1.

Presentation Layer Simple Java GUI
Business Layer Java bean and other objects
Integration Layer JDBC, Connector
Resource Layer Database

Table 4.1: layers pattern in project

In this project, we use MySQL (a open source database) as our backend database. The detailed steps of setting up MySQL can be found in Appendix A.

4.3 Transaction management

4.3.1 Distributed transaction concept

In a peer-to-peer system several components together provide a certain global behavior. So if one wants to take this global behavior from one correct state to another correct state, one needs to take all participating components within one operation to this new state. To do so we need a transaction that spans multiple components and can commit the actions in all components or abort all these actions.

A distributed transaction, sometimes referred to as a global transaction, is a set of two or more related local transactions that must be managed in a coordinated way. The

transactions that constitute a distributed transaction might be in the same database, but more typically are in different databases and often in different locations. Distributed transactions are very important for the global behavior of distributed Petri nets. In the followings, we first define the concept of transaction:

A transaction can be defined as an indivisible unit of work comprised of several operations, all or none of which must be performed in order to preserve data integrity. There are several different states in a transaction: initial, partially committed, committed and aborted. The partially committed state is a state before the committed state. The transaction states are shown in Figure 4.7.

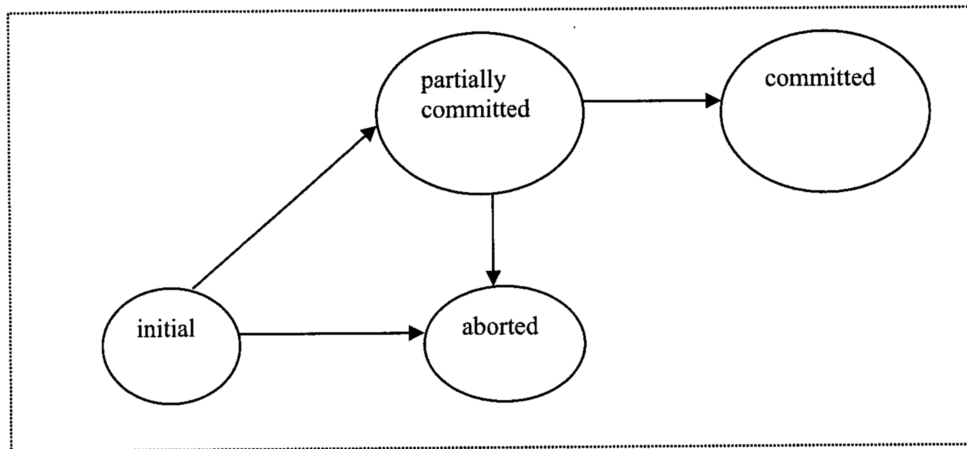


Figure 4.7: Transaction states

The concept of transactions requires system-level software support to make these transactions atomic. A transaction manager is a piece of software that takes care of this by executing a termination protocol that addresses all of the resources that are involved in the transaction.

By definition, a transaction is a unit of work done by multiple distributed components on shared data, with the following properties [36]:

- Atomicity: A transaction should be done or undone completely and unambiguously. That means when there is a failure of any operation, effects of all operations that make up the transaction should be undone, and data should be rolled back to its previous state.
- Consistency: A transaction should preserve all the invariant properties defined on the data. On completion of a successful transaction, the data should be in a consistent state. In other words, a transaction should transform the system from one consistent state to another consistent state.
- Isolation: Each transaction should appear to execute independently of other transactions that may be executing concurrently in the same environment. The effect of executing a set of transactions concurrently should be the same as that of running them serially.
- Durability: The effects of a completed transaction should always be persistent.

These properties, also called ACID properties, guarantee that a transaction is never incomplete, the data is never inconsistent, concurrent transactions are independent, and the effects of a transaction are persistent.

There are two aspects to atomicity:

- All-or-nothing: a transaction either complete successfully and the effect of all of its operations are recorded in the data items or it has no effect at all. So the effects are atomic even when the server fails. After a transaction has completed successfully all its effects are saved in permanent storage.

- Isolation: each transaction must be performed without interference from other transactions.

A typical transaction processing architecture includes a transaction manager to implement transactions and preserve the ACID properties of transactions with the help of resource managers. A resource manager is a component that manages persistent and stable data storage systems.

In order to guarantee that the distributed transaction outcome is atomic, we need the two-phase commit protocol. The two-phase commit protocol works in two phases: a voting phase and a decision phase.

In the first phase, the voting (or prepare) phase, the transaction manager will conduct voting among all the resource managers registered for the transaction in question to determine whether they can agree with a successful termination or not. Each may return a negative reply, for instance if there was an error which caused the transaction to be rolled back. The transaction manager calls prepare method on each resource manager, which will return a ready message if positive. The ready message indicates that the resource manager is ready to commit the operations. In case the resource manager wants the transaction to be rolled back, it sends an abort message.

The second phase is a commit or recover phase. Depending on whether all resource managers are ready for a commit or not, one of these phases will be invoked by the transaction manager.

After the transaction manager has received all of the replies (also called 'votes') it will make a global decision on the outcome of the transaction. This decision will depend on the collected replies: If all the replies were positive, then the transaction manager will

instruct all to commit. If at least one reply is negative (or missing) then a rollback decision is sent to the remaining resources.

The whole process is shown by Figure 4.8.

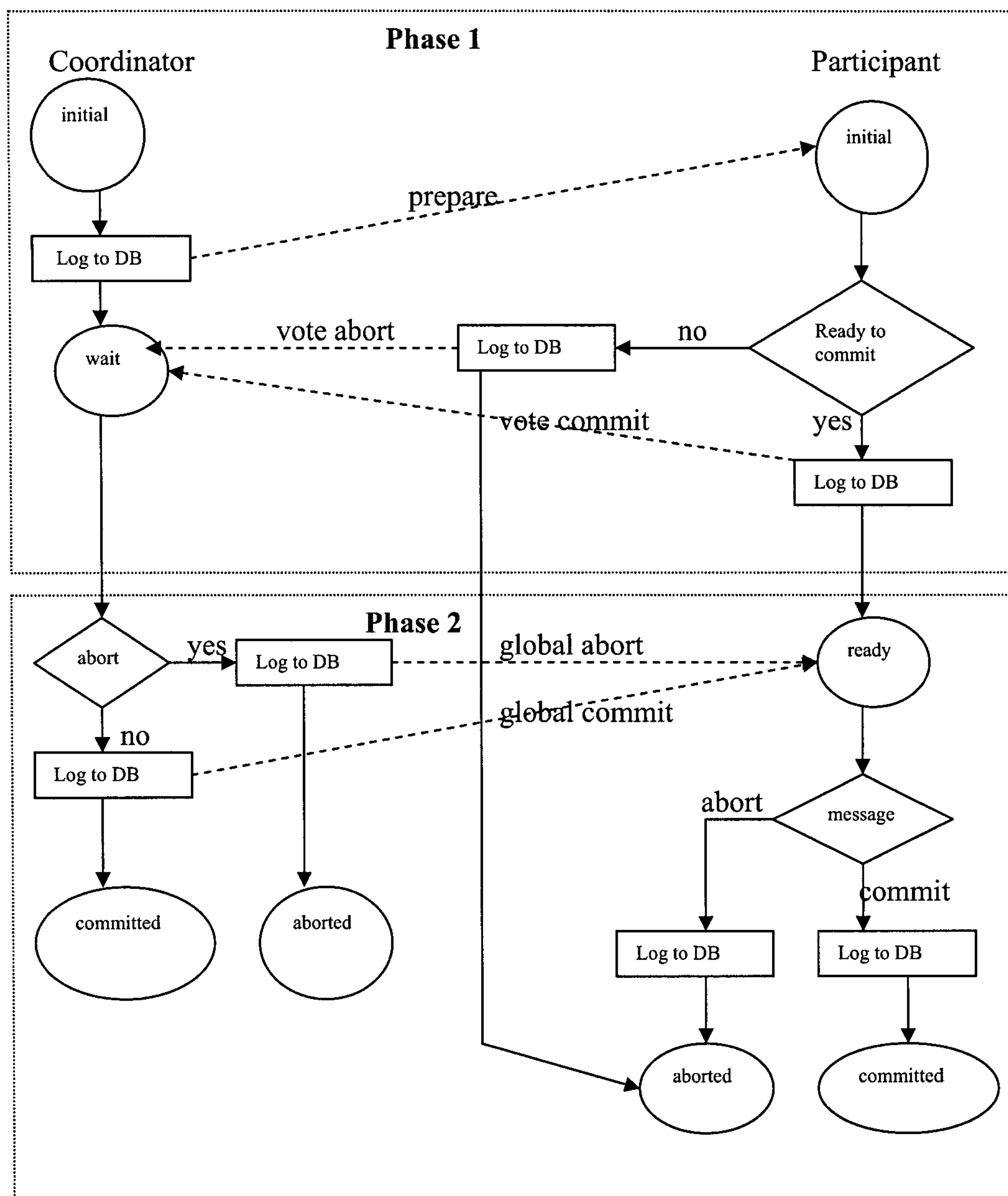


Figure 4.8: Two-phase commit protocol

The circles represent states, the whole lines represent state changes, the dashed lines represent messages (labelled by the message type), and the rectangles represent logging actions. From Figure 4.8, we can see that for a complete and successful transaction:

- The Coordinator will write transaction information at three times:

1. Before sending prepared message
2. Before sending commit message
3. After getting the committed message

- Each participant will write transaction information at two times:
  1. After sending prepared message
  2. After sending committed message

4.3.2 Principles of two-phase commit protocol implementation

In this section, we will look at the implementation of the two-phase commit protocol and recovery issues. The two-phase commit protocol ensures that all transactions are atomic and the databases involved in the transaction have been updated successfully.

Let us look at the two-phase commit protocol implemented between two different nodes (NodeA and NodeB):

1. NodeA saves new transaction data in database before sending a prepared message to NodeB. NodeB will send back a ready message to NodeA as response.
2. NodeA commits and sends a commit message to NodeB. NodeB commits and sends back a message to NodeA as response.

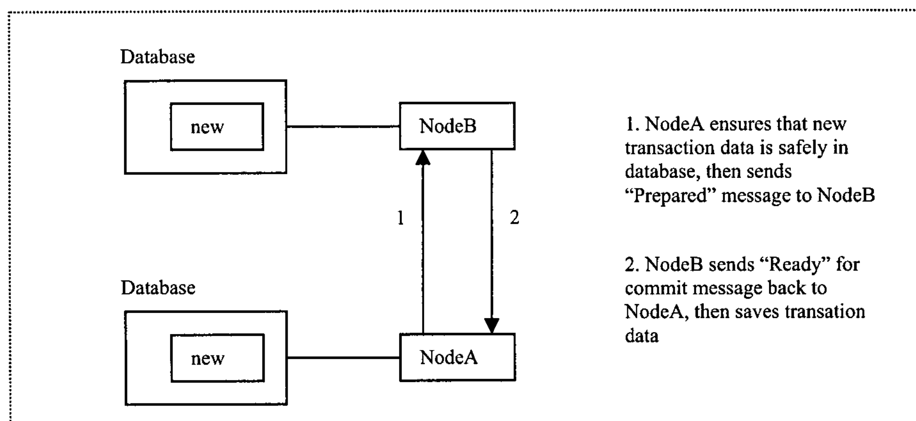


Figure 4.9: First phase

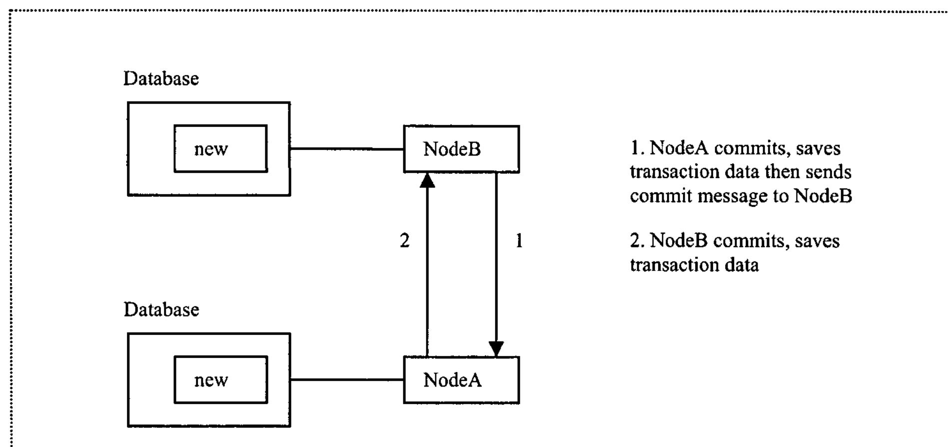


Figure 4.10: Second phase

The two-phase commit protocol is implemented using some general communications protocols such as remote procedure call or some form of message passing. Of course, sometimes there is congestion or some failure of communication protocols. In order to detect this issue, a timeout mechanism is used. If a timeout happens during the two-phase commit process, the transaction will be aborted by the coordinator.

In our example, the NodeA starts the transaction and is the transaction Coordinator; the NodeB is the participant. A RPC-based protocol is used to send and receive messages between the NodeA and the NodeB. MySQL is used to save transaction information.

4.3.3 Transaction recovery

The main purpose of the transaction recovery is to check transaction records, find all uncommitted transactions and resume them.

There are lots of reasons that recovery is needed for atomic transactions:

- System crash
- Hardware failure
- Transaction abort

The transaction failure can occur either in the coordinator or the participant.

As we discussed above, in the normal case, the coordinator starts a distributed transaction, sends messages to each participants to prepare and commit the transaction.

If the coordinator or participant fails, we need the following recovery strategy:

If the coordinator fails before it writes a committed record:

- If coordinator wrote a prepare record before crash, the coordinator will send prepare message to participants again during the recovery.
- If coordinator wrote a commit record before crash, the coordinator will send commit message to participants again during the recovery.

If the participant fails

- before writing the prepared record: It will not vote in the transaction and coordinator will use timeout mechanism to detect it. And the whole transaction will be cancelled.
- after writing the prepared record, before crash: Contact the coordinator how to solve it. For this case, the participant will send a transactionID to the coordinator, then the coordinator will check its database, if it finds the transactionID, then it will read transaction information and restart the transaction; if it cannot find the transactionID in its database, it will send deletion message to delete the transaction message in the participant.

In our implementation, after a participant finds the prepared transaction (not committed), it will delete the transaction information after sending the transactionID to the coordinator. The reasons are when coordinator restarts the transaction, it will send the transaction information to the participant with the same transactionID again, and the participant saves the transaction information in its database. If the coordinator doesn't

want to restart the transaction, it will not send the abort message again in order to reduce network traffic.

The basic approach to achieving crash resilience is logging. Our goal is to make a single invocation atomic in the presence of crashes.

- Logging: Persistent object values and all changes are recorded in a log. The method is based on recorded state with the help of a database.

We require an object to write a log record for all the invocations on it. Depending on the use that is to be made of the log, it might only be necessary to log the invocations that lead to a changed object state.

A log record might take the following general form:

Transaction id, Transaction status, object id, operation, arguments

If a failure of any kind occurs the log can be used in the recovery procedure by the recovery manager.

In our transaction recovery implementation, the recovery manager uses the log record to check whether the transaction is finished or not. If the recovery manager finds an uncommitted transaction in its database, it will restart the transaction again. For the transaction starting from NodeA to NodeB, the NodeA is the coordinator and NodeB is the participant; similarly NodeB is the coordinator and NodeC is the participant. We see that each node has two roles: coordinator and participant.

There are two tables defined in each node: trandatain (incoming data) and trandataout (outcoming data). The trandatain table is used to save incoming transaction information. The trandataout is used to save outgoing transaction information.

The RecoveryManager class will take care of the recovery issues in our recovery implementation. The basic idea is that the class will check the database when the server starts up. If any uncommitted transactions are found, it will check the transaction status and resume it by sending a request to the other site.

```
public class RecoveryManager {
public RecoveryManager(){ ... } // construction method
public boolean Check(){ .. } // check database's transaction information
}
```

Data persistence is the ability to keep data or information around even after a program ends. In our implementation, we use MySQL database to save the transaction information in each node. The sample code below shows the steps to save transaction information to MySQL:

```
dbalayer.conMysql();
dbalayer.addTranDataIn( (String)vect.elementAt(0)+"
Prepared","Prepared",((Client)vect.elementAt(1)).getName(),((Client)vect.elementAt(1)).
getMoney(),
((Task)vect.elementAt(2)).getName(),((Task)vect.elementAt(2)).getCost(),((Task)vect.ele
mentAt(2)).getType());
```

CHAPTER 5: Protocol design and implementation issues

5.1 Overview of issues

A communication protocol defines the sequence of operations and the format of messages to be exchanged among two or more entities. It includes two parts: 1) syntax, i.e., the format of the message to be exchanged and 2) semantic, i.e., the sequence of operations to be performed by each entity. An abstract protocol only contains the sequence of operations.

A protocol implementation must satisfy both parts of the protocol definitions, the syntax and the semantics. In this project, the sequence of operations is defined by the sub-Petri-nets derived from the service specification. The syntax is not defined at this "abstract protocol" level. Different syntaxes can be chosen. We have chosen several versions all based on existing middlewares for distributed applications, such as Java RMI, SOAP, J2EE. Java RMI is a protocol for remote invocation of services for Java and only for Java. This protocol uses a specific protocol syntax based on Java serialization and both sides (client and server) have to be written in Java. SOAP uses an XML-based syntax for interchanging objects. J2EE defines a standard for developing component-based multitier enterprise applications. J2EE uses CORBA for object distribution.

In this thesis, we explore the use of different technologies, such as XML, SOAP, J2EE, Servlets and JMS for the implementation of the syntax and semantics of distributed system defined in terms of an "abstract protocol" in the form of a several sub-Petri nets. Based on these technologies, we have three different approaches to implement these distributed systems: Java RMI based, SOAP based and J2EE based. In this chapter, we

give an explanation of these three implementation approaches, and in Chapter 6, we explain how we have realized these three approaches for the example system introduced in Section 3.3.2.

5.2 Message coding and object serialization

XML stands for eXtensible Markup Language. It has a set of rules to create structured, self-describing documents. These rules define semantic tags. A tag is a piece of text that describes a unit of data, or element in XML. These tags are used to contain and structure data in XML.

XML provides the basis for a wide variety of industry and discipline-specific languages, such as Electronic Business XML (ebXML), and Voice Markup Language (VXML) [37]. XML consists of both tags and content. The uses of XML are rapidly expanding in these years. For example, business partners use XML to exchange data with each other. A lot of tools (JBoss, Apache AXIS, etc) use XML to configure environment settings.

Using XML in application can have such benefits:

- XML is a standard for making documents and messages easy to parse by computers.
- XML platform-independent and extremely flexible.

Let us look at the XML syntax rules:

- All XML elements must have a closing tag and be properly nested. An XML element is a discrete unit of content within a document. It begins and ends with a tag that defines it: e.g. <p>Hi there!</p>
- Tree structure of elements.

- XML attributes are associated with elements and add more semantic information to the element. They are defined like HTML attributes as (name, value) pairs, i.e., <p class="a" id="abc">

An XML document consists of a single element that is called the root element, and it contains all the text and any other elements in the document. For instance, the following XML document includes the root element <address>:

```
<?xml version="1.0"?>
<!-- A well-formed document -->
<address>
        <first-name> David </first-name>
        <last-name> Yang </last-name>
        <street> bell street s </street>
        <city> Ottawa </city>
</address>
```

The grammar rules of defining the legal building blocks of an XML document can be captured in either of two distinct ways: document type definition (DTD) and XML schema definition (XSD).

With the help of grammar rules, we can define the legal building blocks of an XML document. This includes information about where each tag is allowed in the document, and which tags can appear within other tags.

- Document type definition

A document type definition (DTD) is a text file consisting of a set of rules about the structure and content of XML documents. It lists the valid set of elements that may appear in an XML document, including their order and attributes. Let us look at the following address.dtd file for the XML file above:

```
<?xml version="1.0"?>
<!ELEMENT note    (first-name,last-name,street,city)>
```

```
<!ELEMENT first-name (#PCDATA)>
<!ELEMENT last-name (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

All the elements contain text. (#PCDATA stands for parsed character data)

- XML schema definition

XML Schema definition (XSD) provides a much more powerful means for defining XML document structure and its limitations. Using XSD, data constraints, hierarchical relationships, and element namespaces can be specified. The corresponding XML schema file for the above XML file could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="first-name" type="xsd:string"/>
    <xsd:element name="last-name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
</xsd:schema>
```

There is a very good paper [47] about XML syntax and parsing concepts.

Before a document can be validated and used, it must be parsed by XML-aware software. There are two popular approaches for structuring XML parsing software:

- Document Object Model

In the Document Object Model (DOM), the parser will read in an entire XML data source and construct a treelike representation of it in memory. Under DOM, a pointer to the entire document is returned to the calling application. The application can then manipulate the document, rearranging nodes, adding and deleting content as needed.

- Simple API for XML

Simple API for XML (SAX) is an event-based model, which means the parser reads in the XML data source and makes callbacks to its client application whenever it encounters the beginning or end of a distinct section of the XML document.

DOM is easy to use but it is slower than SAX and needs memory for keeping the whole document.

XML's power comes from its flexibility. However, the fact that everyone can define his own tags to describe his data will lead to a problem: If everyone uses the same element names, how can they be distinguished? Namespaces are the answer to this question.

Namespaces distinguish between elements and attributes belonging to different XML applications by providing universally unique names for elements and attributes.

The namespace attribute is placed in the start tag of an element and has the following syntax: xmlns:namespace-prefix="namespace". The following XML file uses a namespace: f.

```
<f:table xmlns:f="http://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

The nameapce is identified by a Uniform Resource Identifier (URI); all child elements with the same prefix are associated with the same namespace.

5.2.1 Introduction to Java object XML serialization

XML documents are text-based. You can view and modify XML documents with a text editor. XML serialization is a great way for applications to maintain state, to read and write configuration files, and to transfer data between processes, applications, and enterprises over a network, including the Internet.

Binary data, however, is not easily read by people and XML provides the portability of data that can be processed on diverse platforms. So we need the data binding between XML and Java objects. The basic idea of data binding is to take an XML document and convert it to an instance of a Java object. Furthermore, that Java class is tailored to a business need and generally matches up with the element and attribute naming in the related XML document.

The process of converting a Java object to a XML document representation is called XML serialization, and the reverse process that converts a XML document into a Java object is called deserialization.

There are lots of products on the market for XML serialization/deserialization of Java objects. They can further be divided into the following two categories:

- Fixed class-XML mapping: These products require configuration before they can be used. This means they use mapping files to map the Java objects and XML documents. These products usually also include a utility for generating classes from an XML schema language, usually DTDs or XML Schemas. Examples are Castor [43], JAXB (Java Architecture for XML Binding) [44], Zeus [45], Apache AXIS [55], etc
- Flexible class-XML mapping: These products do not require any configuration. Instead, they can be used directly in code to serialize and de-serialize objects as XML. In exchange for ease of use, the user generally has no control over how classes are mapped to XML. Example is JSX (Java Serialization to XML) [46].

Apache AXIS uses XML schema definition to define data elements. This schema defines most of the basic data types. The general practice is to declare a namespace identifier

named xsd and associate it with the namespace http://www.w3.org/2001/XMLSchema. Another common namespace identifier used for data encoding is xsi, which is associated with the namespace http://www.w3.org/2001/XMLSchema/instance.

Here is an example:

int value = 11;   <value xsi:type="xsd:int">10</value>

Apache AXIS defines a standard mapping (see Table 5.1) between Java and XML:

xsd:boolean	boolean
xsd:byte	byte
xsd:dateTime	java.util.Calendar
xsd:decimal	java.math.BigDecimal
xsd:double	double
xsd:float	float
xsd:hexBinary	byte[]
xsd:int	int
xsd:integer	java.math.BigInteger
xsd:long	long
xsd:QName	javax.xml.namespace.Qname
xsd:short	short
xsd:string	java.lang.String

Table 5.1: mapping between Java Type and XSD

Apache AXIS also has the ability to serialize/deserialize a Java class that follows standard JavaBean pattern of get/set accessors.

In order to use the bean serialization and deserialization abilities of Apache AXIS, our classes (Worker, Client, Task) follow the standard of JavaBean class pattern. For instance, the class Client contains the following definitions:

```
public  class Client {
        public Client() {};
        public Client( String name, int money)
        {
                this.Name=name;
```

```
        this.Money=money;
    }
    private String  Name ;
    public String getName()
    {
        return Name;
    }
    public void setName(String name)
    {
        Name = name;
        }
    private int Money;
    public int getMoney()
    {
        return Money;
        }
    public void setMoney(int money)
    {
        Money = money;
        }
    }
```

In order to let Apache AXIS serialize/deserialize the Client class, we have to add the JavaBean mapping configuration to nodea.wsdd, a Web Service Deployment Descriptor (WSDD) file that contains all the services that we want to deploy into Apache AXIS.

This configuration file contains:

```
<beanMapping qname="nodea:Client" languageSpecificType="java:Client"/>
```

The <beanMapping> element maps the Java class: Client (presumably a bean) to an XML Qname. QName class represents the values of XML qualified names. The value of a QName contains a namespaceURI and a localPart. The namespaceURI is a URI reference identifying the namespace and the localPart provides the local part of the qualified name. A detailed document about QName can be found at [48].

Let us look at the serialized SOAP message between two sites. Let us assume that an object that resides on site A will call the method RequestFromA of an object that resides

on B site by sending two Java objects (client object, task object). The client object has the following attributes: Name, Money.

The values of the attributes Name is "client1" and of Money is "100".

The task object has the following attribute values: Name, Type and Cost.

The values of these attributes are Name is "task2", Type is "1" and Cost is "10".

This request will result in the following HTTP request containing a SOAP [49] request:

```
POST /axis/servlet/AXISServlet HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: AXIS/1.0
Host: localhost
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: "http://www.simpleprotocol.org/NodeB"
Content-Length: 1214

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:RequestFromA
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="http://www.simpleprotocol.org/NodeB">
   <P11 href="#id0"/>
   <P22 href="#id1"/>
  </ns1:RequestFromA>
  <multiRef id="id1" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xsi:type="ns2:Client"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns2="http://www.simpleprotocol.org/NodeB">
   <clientName xsi:type="xsd:string">client1</clientName>
   <clientMoney xsi:type="xsd:int">100</clientMoney>
  </multiRef>
  <multiRef id="id0" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xsi:type="ns3:Task" xmlns:ns3="http://www.simpleprotocol.org/NodeB"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
```

```
  <taskName xsi:type="xsd:string">task2</taskName>
  <taskType xsi:type="xsd:int">1</taskType>
  <taskCost xsi:type="xsd:int">10</taskCost>
 </multiRef>
</soapenv:Body>
</soapenv:Envelope>
```

There is a href attribute in the SOAP message. The href attribute in the SOAP message is of type uri-reference defined in the XML Schema specification; the value of this attribute references the id attribute of the element that contains the data value.

We can see that in this SOAP message, there are two references: id0 and id1. The ability to reference the value of other elements is very important. First, it may reduce the message size. Second, it is also useful when serializing a graph, or collection, of objects where many of those objects have references to the same object. In this sample message above, no value is provided to these references; instead, an attribute identifies the element where the actual data value is provided after these references.

For our implementation, Apache AXIS can serialize/deserialize our JavaBean classes. When the bean serialization model of Apache AXIS is not enough to handle complex data type, we can use custom serialization. That means we need to write serialization and deserialization Java methods for the complex data type.

5.3 Introduction to RPC-based technologies

5.3.1 Introduction to SOAP

In recent years, the easy access to the Internet and the rapid growth of the number of servers connected to it have made the World Wide Web and private Intranets the natural candidate for developing and deploying distributed applications. There are lots of solutions for distributed computing, such as Microsoft DCOM, CORBA, Java RMI, etc.

In order to increase network security, firewalls are widely used in the world. A computer firewall protects networked computers from intentional hostile intrusion that could compromise confidentiality or result in data corruption or denial of service. A firewall may be a hardware device or a software program running on a secure host computer.

But previous solutions (Microsoft DCOM, CORBA, Java RMI) for distributed computing were not good for interoperability through firewall. They are very good in a local network. But these protocols show their limitations when they are deployed over the Internet. Because the Internet has lots of firewalls, all these protocols cannot pass these firewalls, and it is not easy to let these protocols talk to each other easily. We need a new protocol that can send and receive messages via HTTP and solve all these limitations.

SOAP (Simple Object Access Protocol) was designed as "a lightweight protocol for information exchange in a decentralized, distributed environment" [49]. SOAP was developed by the W3C (World Wide Web Consortium). It uses the approach of expressing data as XML and transporting it across the Internet using HTTP.

The help of XML for data encoding gives SOAP some unique capabilities: It is much easier to monitor SOAP messages than a binary stream.

SOAP relies on HTTP as a transport mechanism to send XML based messages; the messages are packed in what is called a SOAP envelope and sent to the server to be processed in a Request/Response fashion.

A SOAP message consists of two parts:

1. The Header (optional) that contains additional, application specific information about the SOAP message.
2. The Body contains call of response information.

Let us look at the template of a SOAP message:

```
<SOAP-ENV:Envelope>
        <SOAP-ENV:Header>SOAP Header</SOAP-ENV:Header>
        <SOAP-ENV:Body>
        <My:Info > My Info</My:Info>
        <Your:Info > Your Info</Your:Info>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP header may have two attributes:

1. SOAP-ENV:actor

   A SOAP message travels from the originator to the ultimate destination, potentially by passing through a set of SOAP intermediaries along the message path. An actor in SOAP is a different message receiver that may receive the message, and possibly modify it, before forwarding it to either the next actor or the final intended receiver.

2. SOAP-ENV:mustUnderstand

   This attribute indicates how to process the header entry. The receiver may or may not understand and obey the semantics of the message header. If the sender wants to require the receiver understand the message header. It may add a "mustUnderstand=true" attributed to the message header block.

   The SOAP body contains the RPC information.

The SOAP-ENV:Fault element is used to provide information about errors that occurred while processing the message. By nature this element can only appear in answers (response messages). Its sub-elements are

- faultcode: a namespace-qualified fault name
- faultstring: a human-readable string
- faultactor: what actor raised the fault

SOAP defines a set of basic errors:

- VersionMismatch: the SOAP envelope is using an invalid namespace for the SOAP Envelope element.
- MustUnderstand: a header block contained a mustUnderstand="true" flag and some part of the body message.
- Client: there is a problem in the message. For example, the message contains invalid authentication credentials
- Server: an error occurred that cannot be directly linked to the processing of the message.

5.3.2 Introduction to Apache AXIS

Apache AXIS is an open-source implementation of the SOAP submission to W3C; it is a follow-on to the Apache SOAP project (old version of SOAP). Apache AXIS implements the Java API for XML-Based RPC (JAX-RPC) API [50], one of the standard ways to program Java web services.

Compare with Apache SOAP, Apache AXIS provides the following advantages:

- Resource efficient: Apache AXIS uses SAX and Apache SOAP uses DOM. SAX allows lazy parsing of XML documents when the entire XML document does not need to be parsed or kept in memory.
- Extensibility: Apache AXIS has facilitates for inserting of new extensions into the engine for custom header processing and system management. Apache AXIS provides a deployment descriptor for describing various components like services, handler objects, serializers/deserializers, and so on.

- Component-oriented deployment: One can define reusable networks of handlers to implement common patterns of processing for applications.
- Transport framework - AXIS provides a transport framework by providing senders and listeners for SOAP over various protocols.
- WSDL support - AXIS supports the Web Service Description Language (WSDL) [42]. This is an XML schema for describing a web service as a collection of access endpoints capable of exchanging messages in a procedure-oriented or document-oriented fashion.

Let us look at the architecture of Apache AXIS: There are several subsystems in Apache AXIS: admin, service, provider, transport, etc. They work together and provide flexible service. The most important concepts in the message flow subsystem are handler and chain.

The handlers in Apache AXIS are invoked in sequence to process messages. The chain is a handler that may consist of a sequence of handlers and the handlers are invoked in turn. The particular order is determined by the deployment configuration and whether the site is server or client.

When a client makes a call to a server, the client will construct a call object and invoke the AXIS Client engine. When the server gets the call, it will invoke the corresponding Servlet (see [26]) to process the call.

Three WSDD (see [56]) files (nodea.wsdd, nodeb.wsdd, nodec.wsdd) are used to define three different services. These services are used to process incoming PRC call and return corresponding response.

5.3.3 Introduction to Java RMI

Java's Remote Method Invocation (RMI) facilitates object method calls between objects that reside in different Java Virtual Machines (JVMs). JVMs may be located on separate computers; yet one JVM may invoke methods belonging to an object stored in another JVM. Methods can even pass objects of new classes that a foreign virtual machine has never encountered before, allowing dynamic loading of new classes definition as required [29].

Java RMI is realized in the three layers, as shown in Figure 5.1.

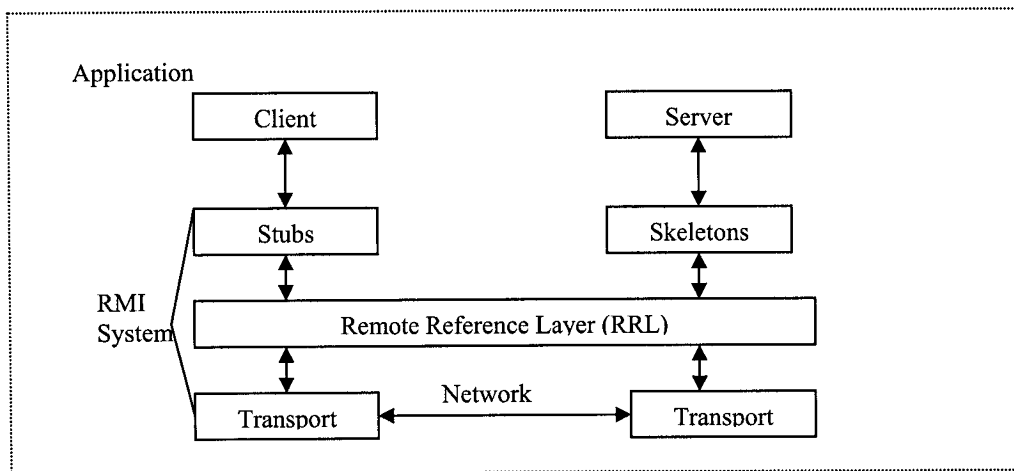


Figure 5.1, Java RMI structure

- The first layer is the Stub/Skeleton Layer. This layer is responsible for managing the remote object interface between the client and server.
- The second layer is the Remote Reference Layer (RRL). This layer is responsible for managing the "liveliness" of the remote objects. It also manages the communication between the client/server and virtual machine s, (e.g., threading, garbage collection, etc.) for remote objects.
- The third layer is the transport layer. This is the actual network/communication layer that is used to send the information between the client and server over the wire. It is normally based on TCP/IP.

There are two kinds of classes that can be used in Java RMI.

1. A remote class is one whose instances can be called remotely. An object of such a class can be referenced in two different ways:

- Within the address space where the object was constructed, the object is an ordinary object that can be used like any other object.
- Within other address spaces, the object can be referenced using a stub. While there are limitations on how one can use a stub compared to an object, for the most part one can use stubs, also called object handles in the same way as an ordinary object.

2. A serializable class is one whose instances can be copied from one address space to another. An instance of a serializable class will be called a serializable object. If a serializable object is passed as a parameter (or return value) of a remote method invocation, then the value of the object will be copied from one address space to the other. By contrast if a remote object is passed as a parameter (or return value), then a stub will be copied from one address space to the other.

Java's binary serialization API (whose major classes are ObjectOutputStream and ObjectInputStream) provides an infrastructure that supports data serialization into binary form.

The whole calling procedure is shown in Figure 5.2.

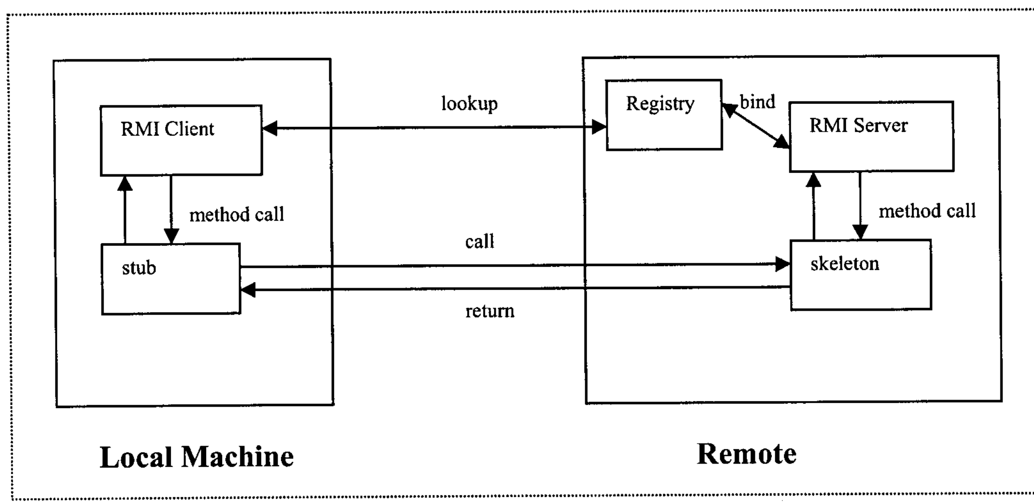


Figure 5.2: Java RMI calling procedure

The server object must first register itself with a name in the registry, and the client looks up the server name in the registry to obtain a remote reference in the form of a stub. When a method is called on the stub, the stub can serialize the parameters and send the method call to the skeleton, the skeleton invokes the method on the server object and sends the result back to the stub.

It is important to note that a client may also obtain a stub pointing to a server as return values of method calls from other objects independently from the registry.

5.4 Introduction to J2EE

The current trend in enterprise software development is to provide multiple-tier frameworks aimed at delivering applications that are secure, scalable, and available. For example, a 3-tier framework is shown in Figure 5.3.

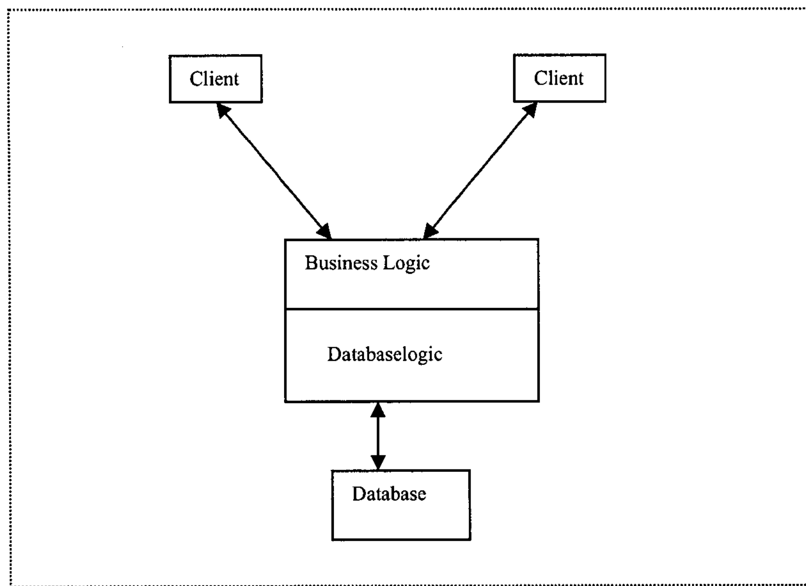


Figure 5.3: 3-tier framework

To this end, Sun Microsystems introduced Java 2 Enterprise Edition (J2EE), and Microsoft Corporation ventured the .NET framework to help developers build applications that are Web-friendly and easily used to deliver e-commerce solutions [27].

J2EE was defined by SUN for producing secure, scalable, and highly available enterprise applications. This standard defines which services should be provided by application servers that support J2EE. These servers will provide J2EE containers in which J2EE components will run.

The typical three-tier J2EE environment is shown in Figure 5.4.

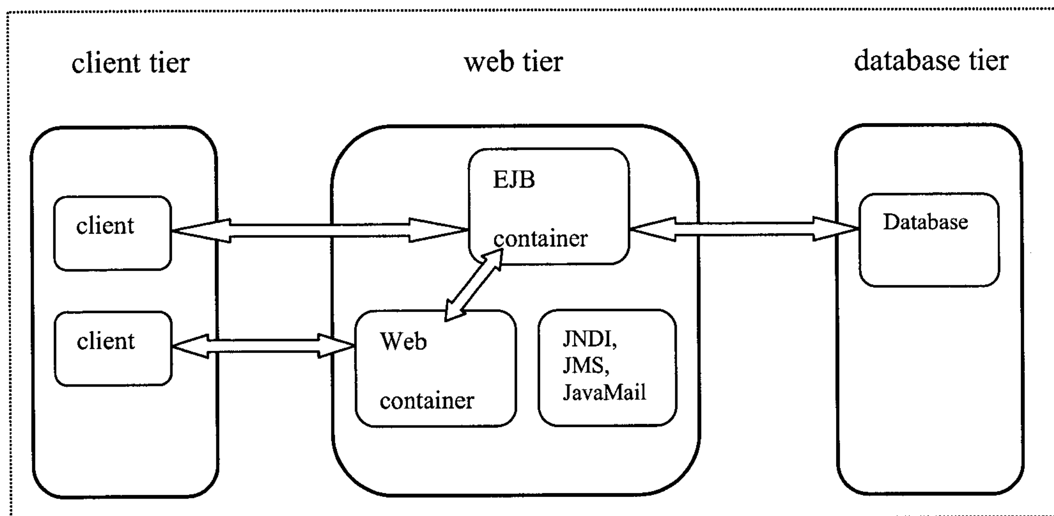


Figure 5.4: J2EE three-tier architecture

The J2EE client tier provides support for a variety of client types, both within the enterprise firewall and outside. Clients can be offered through Web browsers by using HTML pages generated with the JavaServer Pages (JSP) technology, or by Servlets. Clients can also be offered as stand-alone Java language applications.

The J2EE middle tier supports client services through web containers in the web tier and supports business logic component services through Enterprise JavaBeans (EJB) containers in the EJB tier. J2EE specification allows the component or application developer to concentrate on the business logic while the complexities of delivering a reliable, scalable service are handled by the EJB server. The EJB component model is the backbone of the J2EE programming model, while we will discuss in the following section.

The database tier supports access to existing database systems by using Java DataBase Connectivity (JDBC) API [51].

The main benefits of the J2EE platform are:

- Simplified architecture and development: the J2EE application model provides the benefits of Write Once, Run Anywhere portability and scalability for multi-

tier applications. This standard model minimizes the cost of developer training while providing the enterprise with a broad choice of J2EE servers and development tools.

- Easy to use: J2EE uses XML deployment descriptors to manage the operations of the server. It is very straightforward. With the help of container-managed persistence, there is no need to write any JDBC code. It also provides transaction management.

5.4.1 Introduction to EJB

A Java Enterprise Bean (EJB) is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application.

Several benefits of using EJB:

- EJB simplify the development of large, distributed applications. First, because the EJB container provides system-level services (naming, transaction, etc) to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container—not the bean developer—is responsible for system-level services such as transaction management and security authorization.
- Because the beans—and not the clients—contain the application's business logic, the client developer can focus on the presentation to the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

- Because enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant J2EE server.

You should consider using enterprise beans if your application has any of the following requirements:

- The application is scalable. To accommodate a growing number of users, you may need to distribute an application's components across multiple machines. Not only can the enterprise beans of an application run on different machines, but their location will remain transparent to the clients.
- Transactions are required to ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access to shared objects.
- The application will have a variety of clients. With just a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

There are three types of EJB:

- Session bean: session beans are used to implement business objects that hold client-specific business logic. The state of such a business object reflects its interaction with a particular client and is not intended for general access. Therefore, a session bean typically executes on behalf of a single client and cannot be shared among multiple clients. When a new client references a session bean from the server, the EJB container creates a new instance of the session bean, which is tied to the client. In contrast to entity beans, session beans do not directly represent shared data in the database, although they can access and update such

data. The state of a session object is non-persistent and need not be written to the database. There are two types of session beans: stateful session bean and stateless session bean.

- Entity bean: an entity bean represents an object view of business data stored in persistent storage or an existing application. The bean provides an object wrapper around the data to handle the data persistence and transaction management. This object interface lends itself to software reuse. An entity bean allows shared access from multiple clients and can live past the duration of client's session with the server.
- Message-driven bean: a message-driven bean is an enterprise bean that allows J2EE applications to process messages asynchronously. It acts as a JMS (Java Message Service) message listener, which is similar to an event listener except that it receives messages instead of events.

The client view of an enterprise bean is provided through two interfaces. These interfaces are implemented by classes constructed by the container when a bean is deployed, based on the information provided by the bean.

The home interface provides methods for creating and removing enterprise beans. This interface must extend javax.EJB.EJBHome.

The remote interface defines the client view of an enterprise bean—the set of business methods available to the clients. This interface must extend javax.ejb.EJBObject.

The enterprise bean class is the second part of the mechanism that allows for container-managed services in the EJB architecture. It provides the actual implementation of the business methods of the bean. It is called by the container when the client calls the

corresponding methods listed in the remote interface. This class must implement the javax.ejb.EntityBean or javax.ejb.SessionBean interface.

5.4.2 Introduction to JMS

Enterprise messaging systems allow two or more applications to exchange information in the form of messages. A message, in this case, is a self-contained package of business data.

A key concept of enterprise messaging is messages are delivered asynchronously from one system to others over a network. To deliver a message asynchronously means the sender is not required to wait for the message to be received or handled by the recipient; it is free to send the message and continue processing.

The Java Message Service (JMS) is a Java API developed by SUN Microsystems that can be used with many different Message-Oriented Middleware vendors. JMS is analogous to JDBC in that application developers reuse the same API to access many different systems. If a vendor provides a compliant service provider for JMS, then the JMS API can be used to send and receive messages to that vendor [28].

JMS provides for two types of messaging models, publish-and-subscribe and point to point queuing. In JMS terminology, publish-and-subscribe (See Figure 5.5) and point-to-point (See Figure 5.6) are frequently shortened to pub/sub and ptp, respectively.

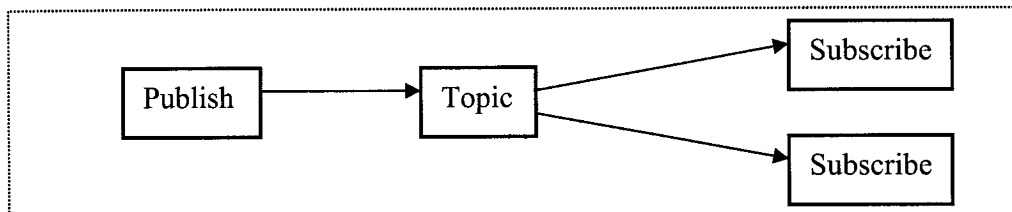


Figure 5.5: JMS publish-and-subscribe model

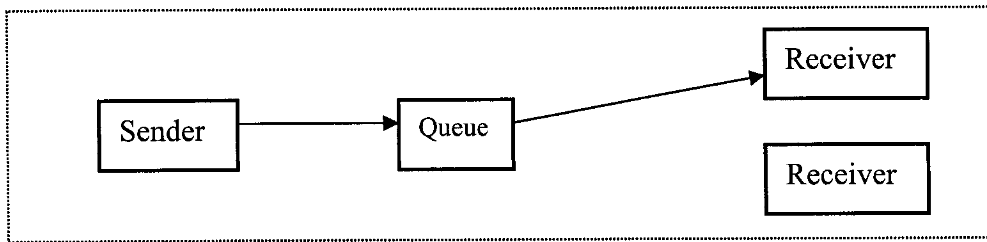


Figure 5.6: JMS point-to-point model

In the publish-and-subscribe model, one producer can send a message to many consumers through a virtual channel called a topic. Consumers, which receive messages, can choose to subscribe to a topic. Any messages addressed to a topic are delivered to all the topic's consumers. Every consumer receives a copy of each message. The pub/sub messaging model is by and large a push-based model, where messages are automatically broadcast to consumers without them having to request or poll the topic for new messages. In the model the producer sending the message is not dependent on the consumers receiving the message. Optionally, JMS clients that use pub/sub can establish durable subscriptions that allow consumers to disconnect and later reconnect and collect messages that were published while they were disconnected.

The point-to-point messaging model allows JMS clients to send and receive messages both synchronously and asynchronously via virtual channels known as queues. The ptp messaging model has traditionally been a pull-based or polling-based model, where messages are requested from the queue instead of being pushed to the client automatically. In JMS, however, an option exists that allows ptp clients to use a push model similar to pub/sub.

A given queue may have multiple receivers, but only one receiver may consume each message. As shown in Figure 5.6, the JMS provider takes care of the work, insuring that each message is consumed once and only once by the next available receiver in the group.

The JMS specification does not dictate the rules for distributing messages among multiple receivers, although some JMS vendors have chosen to implement this as a load balancing capability. Ptp also offers other features, such as a queue browser that allows a client to view the contents of a queue prior to consuming its messages – this browser concept is not available in the publish-and-subscribe model.

5.5 Introduction to Servlets

The Servlet specification defines a server-side component model that can be implemented by an application server. Servlets provide a simple but powerful API for generating Web pages dynamically. (Although Servlets can be used for many different request-reply protocols, they are predominantly used to process HTTP requests for web pages.)

Servlets are developed in the same fashion as enterprise session beans; they are Java classes that extend a base component class and may have a deployment descriptor. Servlets do not implicitly support transactions and are not accessed as distributed objects. A Servlet is created to respond to a request, usually HTTP, and processes the respond in the form of an output stream. Once a Servlet is developed and packaged in a JAR file, it can be deployed in an application server. When a Servlet class is deployed, it is assigned to handle requests for a specific web page or assist other Servlets in handling page requests [26].

5.6 Introduction to code generation

Writing programs that write programs is code generation. With today's complex code-intensive frameworks, such as J2EE, Microsoft's .Net, it is very useful to build programs that help us to build applications. Code generation techniques can provide the following benefits [59]:

- Quality: Code generation using templates creates a consistent code base. Templates make the coding improvements consistent throughout the code base.
- Consistency: The code that is built by a code generator is consistent with the design of the APIs.
- More design time: With code generation, engineers can have more design time to rewrite the templates and run the generator to produce the fixed code.

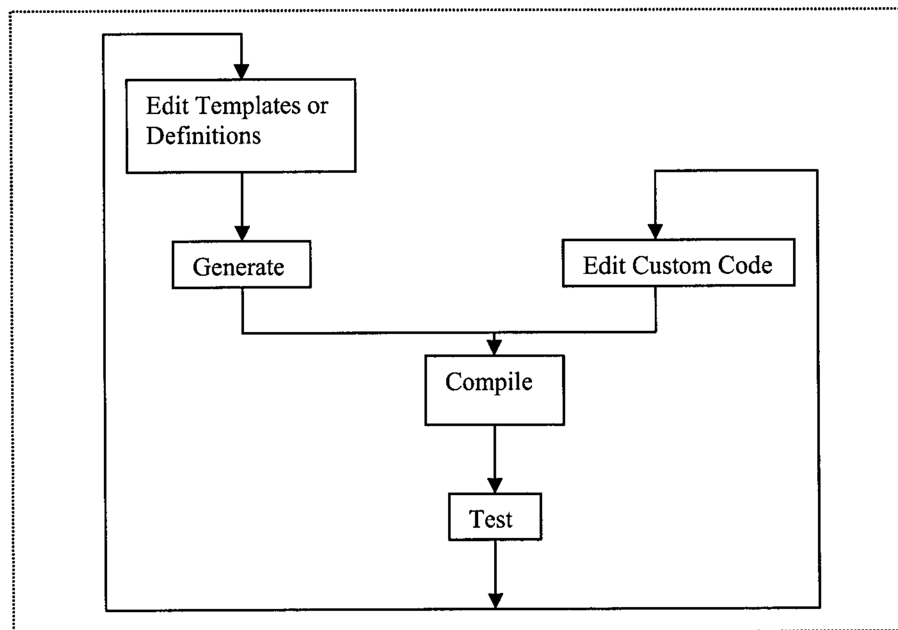


Figure 5.7: Code generation workflow

The Figure 5.7 shows the workflow for the code generation. First, we edit templates and definition files, and then run the generator to generate the output files. All the output files are compiled along with custom code and the application is tested. The text template separates the code formatting logic from the code definition logic.

5.7 Conclusion

Protocol defines the sequences of operations and the format of message. In this chapter, we mainly discussed the technologies used to implement our protocol example. In the

next chapter, we will discuss the three different solutions for our protocol specification example in detail.

CHAPTER 6 Example implementations and comparison

In this chapter, we will talk about three different implementations of the example system described in Section 3.2.2. They are realized using the Java RMI based approach, SOAP based approach and J2EE based approach (See Chapter 5). The different implementations are described in Section 6.1, 6.2 and 6.3, respectively.

For the development of each implementation, we have the following common steps:

1. Define the remote interface of each processing node.
2. Define and deploy services on each processing node.
3. Define the database layer.
4. Test the system.

The issues related to these different steps are further elaborated in the following for each of the implementation approaches.

6.1 Java RMI based approach

The Java RMI architecture is already explained in Section 5.3.3. Here, we will discuss the implementation of the protocol specification using Java RMI. In the following, we explain the four steps above in more detail:

6.1.1 Define the remote interface of each processing node

The remote interface is used for a client to invoke a method on a remote server. All the three different approaches have the same interface definitions. For the call from NodeA to NodeB (see Chapter 3, Figure 3.4), the NodeA is the client and NodeB is the server.

Processing NodeA has the following remote interface definition:

```
public interface NodeARMIInterface extends java.rmi.Remote {
  public void RequestToB( String input) throws java.rmi.RemoteException;
  public String RequestFromB( Vector vec) throws java.rmi.RemoteException;
```

```
  public String RequestFromC( Vector vec) throws java.rmi.RemoteException;
  public  String CommitFromB( String tranid) throws java.rmi.RemoteException;
  public  String CommitFromC( String tranid) throws java.rmi.RemoteException;
}
```

The function: RequestFromB is used by nodeb to send a vector to nodea and the function CommitFromB is used by nodeb to send a commit message to nodeb.

Processing NodeB has the following remote interface:

```
public interface NodeBRMIInterface extends java.rmi.Remote {
  public  String RequestFromA( Vector vect ) throws java.rmi.RemoteException;
  public  String CommitFromA( String tranid ) throws java.rmi.RemoteException;
  public  String RequestFromC( Vector vect ) throws java.rmi.RemoteException;
  public  String CommitFromC( String tranid ) throws java.rmi.RemoteException;
}
```

Processing NodeC has the following remote interface:

```
public interface NodeCRMIInterface extends java.rmi.Remote {
    public  String RequestFromB1( Vector vect) throws java.rmi.RemoteException;
    public  String RequestFromB2 (Vector vect) throws java.rmi.RemoteException;
    public  String CommitFromB1(String tranid) throws java.rmi.RemoteException;
    public  String CommitFromB2 (String tranid) throws java.rmi.RemoteException;
}
```

We can see that the remote interface of each processing node gives the definition of method that can be accessed by other nodes.

6.1.2 Develop the remote object by implementing the remote interface

The remote object extends UnicastRemoteObject class. The UnicastRemoteObject class provides support for creating and exporting remote objects.

6.1.3 Define the database access layer

The class DbALayer of NodeA has the functions to connect/disconnect database, add transaction records and delete transaction records, etc.

```
public class DbALayer {
public void conMysql() throws SQLException { ....} // connect to database
public void addTranData (String tranid, String status,String clientname,int
clientmoney, String taskname, int taskcost,int tasktype) throws SQLException
{ .... } // add transaction data to database
........
```

```
}
```

The NodeB and NodeC also have similar classes: DbBLayer class and DbCLayer class, respectively.

Each processing node has a RMI server to listen to remote requests.

```
public class NodeARMIServer {
 public NodeARMIServer() { .. } // construction method
public static void main(String[] argv) { .. } // main method
}
```

All the class diagrams are shown in Appendix G.

6.1.4 System testing

We have the following scenario to test the system:

1. Setup the RMI running environment (see Appendix C).
2. Start the service on each processing node. The system was initialized with the following values:

   Money of Client1 – 160

   Money of Client2 – 100

   Money of Work1 – 0

   Money of Work2 – 0
3. Run the test program. The scenario is clients will pay money to workers to do some tasks.
4. Verify the running result. System has the final values:

   Money of Client1 – 0

   Money of Client2 – 0

   The total money that Work1 and Work2 together have should be 160.

We can see the screen shots of the three nodes (NodeA, NodeB and NodeC) for a particular execution in Figure 6.1 through 6.3:

Prepared transaction

0 Mon Oct 20 12:29:49 2003 NodeA Prepared,Prepared,client1,160,task2,10,1
1 Mon Oct 20 12:29:49 2003 NodeA Prepared,Prepared,client2,100,task2,10,1
2 Mon Oct 20 12:29:50 2003 NodeA Prepared,Prepared,client1,150,task2,10,1
3 Mon Oct 20 12:29:50 2003 NodeA Prepared,Prepared,client2,90,task2,10,1
4 Mon Oct 20 12:29:51 2003 NodeA Prepared,Prepared,client1,140,task1,20,0

NodeA

Committed transaction

0 Mon Oct 20 12:29:49 2003 NodeA Committed,Committed,client1,160,task2,10,1
1 Mon Oct 20 12:29:49 2003 NodeA Committed,Committed,client2,100,task2,10,1
2 Mon Oct 20 12:29:50 2003 NodeA Committed,Committed,client1,150,task2,10,1
3 Mon Oct 20 12:29:50 2003 NodeA Committed,Committed,client2,90,task2,10,1
4 Mon Oct 20 12:29:51 2003 NodeA Committed,Committed,client1,140,task1,20,0

Client1 Money: 0

Client2 Money: 0

Java Applet Window

Figure6.1: NodeA local transaction records

Prepared transaction

0 Mon Oct 20 12:07:15 2003 NodeB Prepared,Prepared,NU
1 Mon Oct 20 12:07:15 2003 NodeB Prepared,Prepared,cli
3 Mon Oct 20 12:07:16 2003 NodeB Prepared,Prepared,NU
2 Mon Oct 20 12:07:16 2003 NodeB Prepared,Prepared,cli

NodeB

Committed transaction

0 Mon Oct 20 12:07:15 2003 NodeB Committed,Committed,
1 Mon Oct 20 12:07:15 2003 NodeB Committed,Committed,
3 Mon Oct 20 12:07:16 2003 NodeB Committed,Committed,
2 Mon Oct 20 12:07:16 2003 NodeB Committed,Committed,

Worker1 Money: 110

Worker2 Money: 150

Java Applet Window

Figure 6.2: NodeB local transaction records

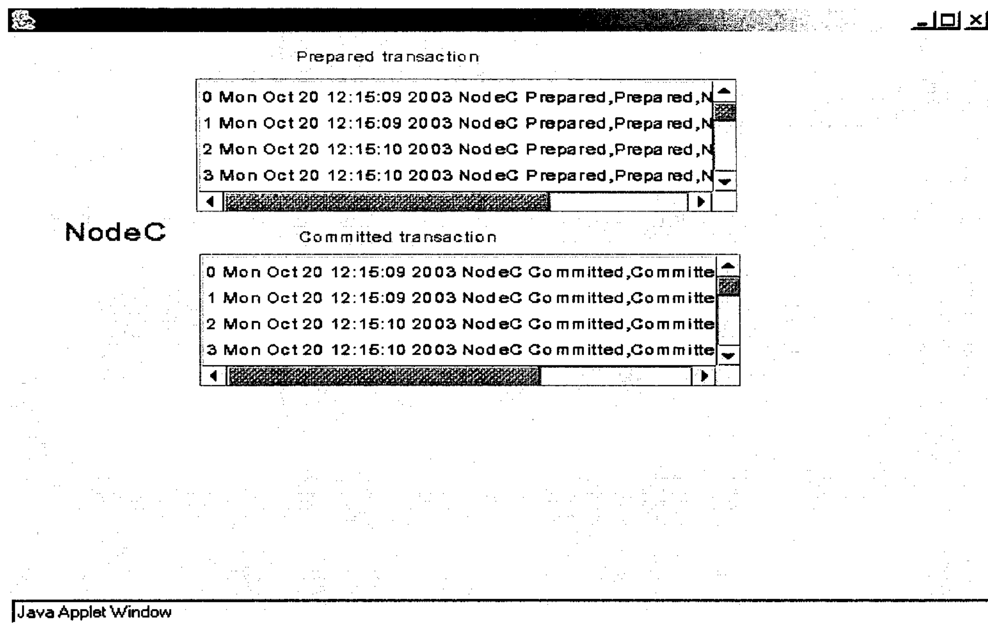


Figure 6.3: NodeC local transaction records

After running, Worker1 has the money 110 units of money and Worker2 has 150. So we can see that the result is right.

6.2 SOAP based approach

In the previous section, we discussed the Java RMI based approach. Compared with the Java RMI approach, the SOAP based approach also has the same interface definitions but uses a different message format: The message format of Java RMI is a binary stream; the message format of SOAP is coded in XML. As we discussed before, the SOAP based approach is slower than Java RMI based approach, but it is firewall-friendly (This is very important for web-based applications).

In the SOAP based approach, we use Apache AXIS to setup XML-based web services. Three different services (NodeAService, NodeBService, NodeCService) can be run on the same machine with three different ports, e.g. 8080, 8081, 8082, or can be run in three

different machines using the same port number, e.g. 8080. The web service of each node is described by a Web Service Description Descriptor (WSDD). The WSDD describes how the various components installed in the AXIS server are to be 'chained' together to process incoming and outgoing messages to the service. These chain definitions are 'compiled' and made available at runtime through registries [31]. The four-implementation steps are described as follows:

6.2.1 Define the remote interface of each processing node

We use the same interface definitions as Java RMI based approach.

6.2.2 Define and deploy services

In each processing node, there is a web service deployment descriptor (WSDD) file.

The nodea WSDD file is shown as follows:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
 xmlns:nodea="http://www.simpleprotocol.org/NodeA"
xmlns:nodeb="http://www.simpleprotocol.org/NodeB"
xmlns:nodec="http://www.simpleprotocol.org/NodeC"
xmlns:reg="http://www.simpleprotocol.org/Registry">
   <service name="NodeA" provider="java:RPC">
      <namespace>http://www.simpleprotocol.org/NodeA</namespace>
      <parameter name="className" value="NodeAService"/>
      <parameter name="allowedMethods" value="RequestToB RequestFromB
RequestFromC CommitFromB CommitFromC"/>
      <parameter name="scope" value="application"/>
   </service>
   <beanMapping qname="nodea:Task" languageSpecificType="java:Task"/>
   <beanMapping qname="nodea:Client" languageSpecificType="java:Client"/>
   <beanMapping qname="nodea:Worker" languageSpecificType="java:Worker"/>
   <beanMapping qname="nodeb:Task" languageSpecificType="java:Task"/>
   <beanMapping qname="nodeb:Client" languageSpecificType="java:Client"/>
   <beanMapping qname="nodeb:Worker" languageSpecificType="java:Worker"/>
   <beanMapping qname="nodec:Task" languageSpecificType="java:Task"/>
   <beanMapping qname="nodec:Client" languageSpecificType="java:Client"/>
   <beanMapping qname="nodec:Worker" languageSpecificType="java:Worker"/>
</deployment>
```

The parameter: className indicates the corresponding Java class (NodeAService) for the service (NodeA).

The parameter: allowedMethods defines all the methods (RequestToB RequestFromB RequestFromC CommitFromB CommitFromC) allowed in the processing nodea.

The parameter: scope indicates this is an application.

The beanMapping is used to map Java bean classes and SOAP qnames.

6.2.3 Define the database access layer

The step is same as Java RMI based approach.

6.2.4 System testing

We use same test scenario as Java RMI approach.

The Apache AXIS running environment setup can be found in the Appendix B.

We can see the screen shots of three processing nodes (NodeA, NodeB, NodeC) for a particular run Figures 6.4 through 6.6.

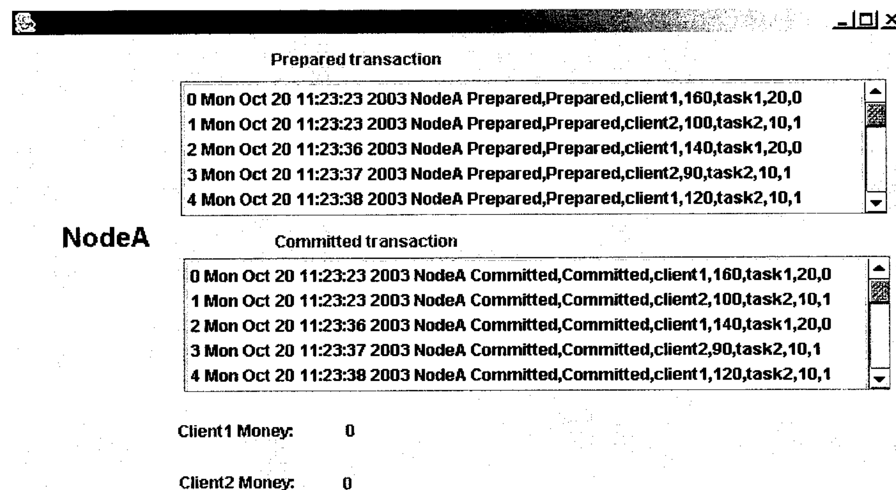


Figure 6.4: NodeA transaction records

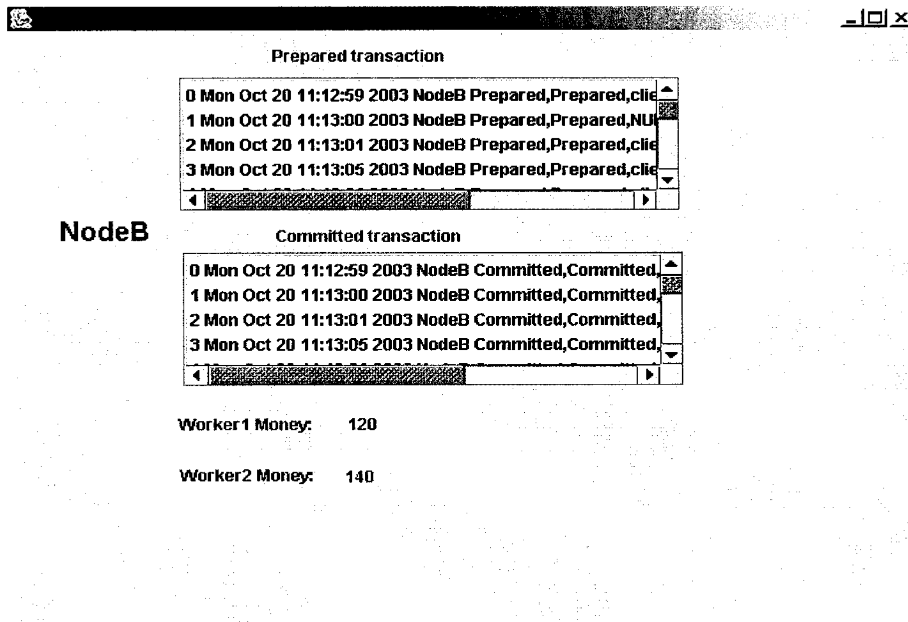


Figure 6.5: NodeB transaction records

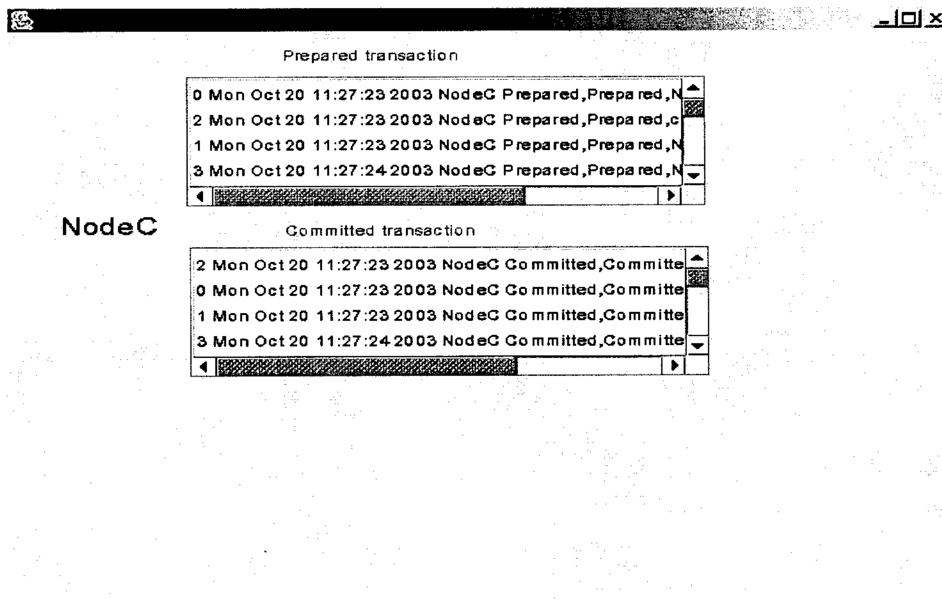


Figure 6.6: NodeC transaction records

The total amount of money that client1 and client2 have at the end is $260.

From the screen shot of NodeB, we can see that worker1 has initially 20 dollars and worker2 has initially 140 dollars. So the running result is correct (120+140=260).

6.3 J2EE based approach

J2EE gives a simplified architecture for rapid development. Compared with the Java RMI based approach and the SOAP based approach, the J2EE based approach is based on EJB which makes it easier to develop and maintain the implementation.

For the J2EE approach, we use the following implementation steps:

6.3.1 Define the remote interface of each processing node

We use same interface definitions as Java RMI based approach.

6.3.2 Define and deploy services on each processing node

According to the J2EE specification, there are two interfaces: EJB home interface, EJB remote interface. For the ClientBean (enterprise Java bean), we have the following code:

```
public abstract class ClientBean implements EntityBean{ // Define entity bean
private EntityContext ctx = null;
public abstract String getName();
public abstract void setName(String name);
public abstract int getMoney() ;
public abstract void setMoney(int money) ;
/**Create an instance of a Client. Note that this method returns null because the real
creation is managed by the EJB container*/


public String ejbCreate (String name, int money){
    setName(name);
    setMoney(money);
    return name;}
public void ejbPostCreate(String name, int money) { }// Called when the object has
//been instantiated;
public void setEntityContext(EntityContext ctx) { this.ctx = ctx; }
public void unsetEntityContext() { ctx = null; }
public void ejbActivate() { }
public void ejbPassivate() { }
public void ejbLoad() { }
public void ejbStore() { }
public void ejbRemove() { }
}

public interface ClientHome extends EJBHome{//Defines the home interface
```

```
public Client create(String name,int money) throws CreateException;// Create a new
//Client instance
/* Find the Client with the specified name. This method is not implemeted by the
Bean,but by the container*/
public Client findByPrimaryKey (String name) throws RemoteException,
FinderException;
/*Get all client instances. This method isimplemented by the container*/
public Collection findAll() throws RemoteException, FinderException;
}

public interface Client extends EJBObject // Defines the remote interface
{
public abstract String getName() throws RemoteException;
public abstract void setName(String name) throws RemoteException;
public abstract int getMoney() throws RemoteException;
public abstract void setMoney(int money) throws RemoteException;
}
```

The ClientHome is the home interface and the Client is the remote interface for ClientBean (see Section 5.4.1).

For the WorkerBean, we have the following code:

```
public abstract class WorkerBean implements EntityBean { // Define entity bean
private EntityContext ctx = null;
public abstract String getName();
public abstract void setName(String name);
public abstract int getMoney() ;
public abstract void setMoney(int money) ;
public String ejbCreate (String name, int money) {
      setName(name);
      setMoney(money);
      return name;  }
public void ejbPostCreate(String name, int money) { }
public void setEntityContext(EntityContext ctx) { this.ctx = ctx; }
public void unsetEntityContext() { ctx = null; }
public void ejbActivate() { }
public void ejbPassivate() { }
public void ejbLoad() { }
public void ejbStore() { }
public void ejbRemove() { } }

public interface Worker extends EJBObject{
public abstract String getName() throws RemoteException;
public abstract void setName(String name) throws RemoteException;
public abstract int getMoney() throws RemoteException;
```

```
public abstract void setMoney(int money) throws RemoteException;}

public interface WorkerHome extends EJBHome {
public Worker create(String name,int money) throws RemoteException, CreateException;
/** Find the Worker with the specified name. This method is not implemeted by the Bean,
but by the container*/
public Worker findByPrimaryKey (String name) throws RemoteException,
FinderException;
/**Get all client instances. This method is implemented by the container*/
public Collection findAll() throws RemoteException, FinderException;}
```

A session enterprise bean models business lgoics and must implement the interface javax.ejb.SessionBean to be a session bean. Entity beans model business objects that need to persist beyond the life of a client instance. Each instance of an entity bean can be accessed by multiple clients simultaneously. Entity beans survive crashes of the server. A bean must implement the interface javax.ejb.EntityBean to be an entity bean. We have to define the session beans (NodeAServiceBean, NodeBServiceBean, NodeCServiceBean) for different service logics running on nodes. In our example, these three session beans (see Section 5.4.1) implement different services for NodeA, NodeB and NodeC.

For the NodeAServiceBean, we have the following code:

```
public interface NodeAService extends EJBObject{ // Define remote interface
public void addClient(String name,int money)
        throws RemoteException, ClientExistsException;
public void modifyClient(String name, int money) throws RemoteException;
public Client findClient(String name) throws RemoteException, FinderException;
public void RequestToB( String input) throws RemoteException;
public  String RequestFromC( Vector vec) throws RemoteException;
public  String RequestFromB( Vector vec) throws RemoteException;
public  String CommitFromC( String tranid) throws RemoteException;
public  String CommitFromB( String tranid) throws RemoteException; }

public class NodeAServiceBean implements SessionBean{//Define NodeAServiceBean
    public static Vector tranAVector = new Vector();
    public static int tranID = 0;
    PlaceA1 placea1;
    PlaceA2 placea2;
    PlaceA3 placea3;
```

```
public static long  timeBefore;
public Properties props ;
public Properties sysProps;
public InitialContext jndiContext ;
private Object getHome (String path, Class type) {...} // return home object
public void addClient(String name, int money)  // add Clientbean
        throws ClientExistsException, RemoteException {...}
public void modifyClient(String name, int money) {...}
public Client findClient(String name) throws RemoteException, FinderException
{...}
 public void RequestToB( String input) {...}
 public  String RequestFromC( Vector vec) {...}
 public  String RequestFromB( Vector vec) {...}
 public  String CommitFromC( String tranid){...}
 public  String CommitFromB( String tranid){...}
public NodeAServiceBean() {...}
public void ejbCreate() {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void setSessionContext(SessionContext sc) {} }

  public interface NodeAServiceHome extends EJBHome // Create home interface
  { NodeAService create() throws RemoteException, CreateException;}
```

The NodeAServiceHome is the home interface and the NodeAService is the remote interface for NodeAServiceBean (see Section 5.4.1).

6.3.3 Define the database access layer

This step is the same as for the Java RMI based approach.

6.3.4 System testing

We use the same test scenario as Java RMI approach.

The class diagrams are shown in Appendix G.

6.4 Performance comparison

In order to compare the performance of these three different approaches, we executed the example program six times.

All three machines ran on Win 2000 and installed the following software:

- Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1)
- Java HotSpot(TM) Client VM (build 1.4.1, mixed mode).
- MySQL version is 4.0.
- Apache AXIS version is 1.0.

All three machines were connected by Local Area Networks(LAN). Each machine's hardware configuration was as follows:

Name	CPU	Ram	Hard driver	Network card
demo1	PII 266 M	256 M	12G	SMC 1211TX
demo2	PIII 450 M	196 M	12G	Realtek RTL8139
Aias103	PIV 1.80 G	256 M	40G	IBM 10/100 Ether

Table 6.1: Hardware configuration

The Java System.currentTimeMillis() method was used for timing measurements. Initially, the system has two workers with zero money and two clients with 260 dollars. The total times for several executions of the test scenario are shown in the tables below:

- For the Apache AXIS approach:

Runing Time (ms)	Worker1 money (dollars)	Worker2 money (dollars)
25176	100	160
22532	170	90
23995	100	160
22843	130	130
24365	130	130
23664	100	160

Table 6.2: AXIS running results

- For the Java RMI approach:

Runing Time (ms)	Worker1 money (dollars)	Worker2 money (dollars)
8171	120	140
7971	110	150
8983	140	120
9684	110	150
8342	160	100
9124	160	100

Table 6.3: Java RMI running results

- For the J2EE approach:

Runing Time (ms)	Worker1 money (dollars)	Worker2 money (dollars)
10775	130	130
10335	180	80
10045	140	120
9513	150	110
9604	130	130
10075	170	90

Table 6.4: J2EE running results

These execution times are also shown in Figure 6.7:

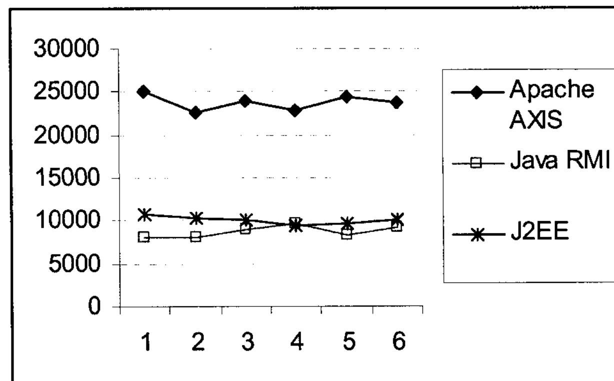


Figure 6.7: AXIS, Java RMI and J2EE performance compared

From Figure 6.7, we can see that Java RMI approach is fastest; J2EE is the second fastest but has a very similar speed. Apache AXIS is the slowest.

We also compared the length of the XML message and Java RMI messages for sending same Java object to the remote system. We got the conclusion: the XML message is almost 1.5 times longer than the RMI message length. Further details about this comparison can be found in Appendix H.

The reason why Apache AXIS is slowest is that AXIS is XML-based. SOAP serialization and deserialization are time consuming.

Because the J2EE architecture uses Java RMI for communication, the performance of J2EE is close to the performance of Java RMI.

Although the performance of SOAP is not good, compared with Java RMI, the interoperability makes it valuable for heterogeneous environments where different systems are written in different programming languages.

CHAPTER 7 Conclusions and future work

7.1 Conclusions

Protocol synthesis focuses on how we can generate a protocol specification derived from a service specification. The problem that we consider here is the solution to implement the protocol specification that is described by distributed Petri nets. We started with the introduction of Petri nets, different protocol synthesis approaches, and then gave the general methodologies to implement distributed Petri nets. Java RMI, SOAP, J2EE, JMS were introduced and used for three different implementations of an example system. A performance comparison was given in the end.

The main contributions of the thesis are:

1. We developed general methodologies for the design of distributed systems. These methodologies can handle highly concurrent control flow as described by Petri nets. The sequential execution task is decomposed into a set of logical processes and each process resides on a different node within a distributed system.
2. Integration within the above methodologies, of persistence and crash recovery using distributed transactions.

The major conclusions from our work are the following:

1. Petri nets have enough modeling and analytical power to specify, verify and analyze large distributed systems, communication protocols and so on.
2. Our general implementation methodologies can handle distributed systems described in terms of Petri nets.
3. For the performance issues, we found that the Java RMI and J2EE approaches have the best performance, while the SOAP approach leads to much less efficient

implementation. The reason is that SOAP messages are in XML format and require much more time for coding and decoding.

7.2 Future work

Deriving a protocol specification by hand is very complex and clearly tool support is desirable. A topic of future work is the development of an integrated development environment for distributed systems with tool support. With the help of such as integrated development, one could use a graphic interface to specify the service specifications, synthesize the protocol specifications and generate the Java codes that implements the protocol specification automatically.

Another point for further study is to investigate different XML parsers for SOAP messages and compare their performance.

Reference

[1] Gregor v. BOCHMANN, Protocol Specification for OSI, Computer Networks and ISDN system, 1990, pp. 167-184.

[2] Robert L. Probert, Kassem Saleh, Synthesis of Communication Protocols: Survey and Assessment, IEEE Transactions on computer, VOL. 40, NO. 4, April 1991, pp. 468-476.

[3] Kassem Saleh, Synthesis of Communications Protocols: An Annotated Bibliography, ACM SIGCOMM Computer Communication Review, Vol. 26, No. 5, 1996, pp. 40-59.

[4] Yamaguchi H., Okano K., Higashino T. and Taniguchi K, Synthesis of Protocol Entities's Specifications from Service Specifications in a Petri Net Model with Registers, IEEE Parallel and Distributed Computing Systems, 1995, pp. 510-517.

[5] Yamaguchi H., Okano K., Higashino T. and Taniguchi K, Protocol Synthesis from Time Petri Net Based Service Specifications, IEEE Parallel and Distributed Systems, 1997, pp. 236-243.

[6] Hakim Kahlouche, Jean Jacques Girardot, A Stepwise Refinement Based Approach for Synthesizing Protocol Specification in an Interpreted Petri Net Model, INFOCOM '96, 1996, pp. 1165-1173.

[7] Merlin P.M., Bochmann G.V., On the construction of submodule specifications and communication protocols, ACM Transactions on Programming Languages and Systems, Vol.5, No. 1, Jan. 1983, pp. 1-25.

[8] Bochmann G.V., Gotzhein R., Deriving protocol specifications from service specifications, Communications, Architectures & Protocols, ACM SIGCOMM' 86, 1986, pp. 144-156.

[9] Chu P.M., Liu M.T., Protocol synthesis in a state-transition model, in Proc. COMPSAC'88, 1988, pp. 505-512.

[10] Saleh K., Probert R.L, Synthesis of error-recoverable protocol specifications from service specifications, Proc. 2nd Intern. Conf. on Computing and Information, Niagara Falls, Canada, May 1990, pp. 428-433.

[11] Zeroual K, Yassini M., A protocol synthesis algorithm: a relational approach, Proc. of the 4th Intl. Conf. on Network Protocols, Nov 1995.

[12] Zhang X.Y., Takahashi K., Shiratori N., Noguchi S., An interactive protocol synthesis algorithm using a global state transition graph, IEEE Transactions on Soft. Eng., Vol. SE-14, No. 3, Mar 1988, pp. 394-404.

[13] P. Zafiropulo et al., Towards analyzing and synthesizing protocols, IEEE Trans. Commun., Vol. COM-28, no. 4, April 1980, pp. 651-661.

[14] D. P. Sidhu, Protocol design rules, in Proc. Second IFIP int. Symp. Protocol Specification, Testing, Verification, 1982, pp. 283-300.

[15] M. G. Gouda, Y. T. Yu, Synthesis of communicating finite state machines with guaranteed progress, IEEE Trans. Commun., Vol. COM-32, no. 7, 1984, pp. 779-788.

[16] Ramamorthy C.V., Dong S.T., Usuda Y., An implementation of an automated protocol synthesizer (APS) and its applications to the X.21 protocol, IEEE Trans. on Software Engineering, Vol. SE-11, No. 9, Sept 1985, pp. 886-908.

[17] Choi T.Y, A Sequence method for protocol construction, in Proc. 6th IFIP International Workshop on Protocol Specification, Testing and Verification, June 1986, pp. 9/1 – 9/18.

[18] Kakuda Y., Wakahara Y., Component-based synthesis of protocols for unlimited number of processes, in Proc. COMPSAC'87, Oct. 1987, pp. 721-730.

[19] Afrati F, et al, The Synthesis of communications protocols, Algorithmica, No. 3, 1988, pp. 451-472.

[20] Kapus-Kolar M., Deriving protocol specification from service specifications including parameters, Microprocessing and Microprogramming, Vol. 32, 1991, pp. 731-738.

[21] Bista B., Cheng Z., Togashi A., Shiratori N., A new approach for protocol synthesis based on LOTOS, IEICE Trans. on Fundamentals, Vol. E77-B, No. 10, October 1994, pp. 1646-1655.

[22] Ando T., Kato Y., Noguchi S., Deriving of communication protocol specification based on Lotos and temporal logic, Proc. of IEEE Singapore Intern. Conf. on Networks, 1993, pp. 591-595.

[23] Chao D, Wand D., The knitting technique and its application to communication protocol synthesis, Proc. of MASCOTS'94.

[24] Manna Z., Wolper P., Synthesis of communicating processes from temporal logic specifications, ACM Trans. on Programming Languages and Systems, Vol. 6, No. 1, Jan 1984, pp. 68-93.

[25] Clarke E.M., Emerson E.A., Synthesis of synchronization skeletons for branching time temporal logic, ACM Trans. on Programming Languages and Systems, Vol. 8, No. 2, March 1986, pp. 244-263.

[26] Marty Hall, Core SERVLETS and JAVASERVER PAGES, Prentice Hall and SUN Microsystems.

[27] Justin Couch, Daniel H. Steinberg, Java 2 Enterprise Edition Bible, 2002, Hungry Minds, Inc.

[28] Richard Monson-Haefel, David A. Chappell, Java Message Service, 2001, O'Reilly.

[29] William Grosso, Java RMI, 2001, O'Reilly.

[30] Khawar Zaman Ahmed, Cary E. Umrysh, Developing Enterprise Java Applications with J2EE and UML, 2001, Addison-Wesley Pub Co.

[31] David Chappell, Tyler Jewell, Java Web Services, 2002, O'Reilly.

[32] Doug Tidwell, James Snell, Pavel Kulchenko, Programming Web Services with SOAP, 2001, O'Reilly.

[33] Mark Matthews, Jim Cole, Joseph D. Gradecki, MySQL and Java Developer's Guide, 2003, Wiley Publishing, Inc.

[34] SCOTT STARK, MARC FLEURY,THE JBOSS GROUP, JBoss Administration and Development, 2002, SAMS.

[35] Murata, T., Petri Nets: Properties, Analysis and Applications, Proc. of IEEE, Vol. 77, No. 4, 1989, pp.541-580.

[36] George Coulouris, Jean Dollimore, Tim Kindberg, Distributed systems concepts and design, second edition, 1998, ADDISON-WESLEY.

[37] KURT A. GABRICK, DAVID B. WEISS, J2EE and XML Development,2002, Manning Publications Co.

[38] Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use. Monographs in Theoretical Computer Science. Berlin, Heidelberg, 1997, New York: Springer-Verlag.

[39] G. Chiola, A. Ferscha, Distributed Simulation of Petri Nets, Parallel & Distributed Technology, Volume: 1 , Issue: 3, Aug. 1993.

[40] H.Yamaguichi, G.V.Bochmann, T.Higashino, Decomposing Service Definition in Predicate / Transition-Nets for Designing distributed Systems, FORTE 2003, pp. 399-414, 2003.

[41] University of Aarhus, Design/CPN, http://www.daimi.au.dk/designCPN, 2005-3-26.

[42] W3C, WSDL, http://www.w3.org/TR/wsdl, 2005-3-26.

[43] Keith Visco, Assaf Arkin, Castor, http://www.castor.org, 2005-3-26.

[44] SUN, JAXB, http://java.sun.com/xml/jaxb/, 2005-3-26.

[45] ObjectWeb, Zeus, http://zeus.objectweb.org, 2005-3-26.

[46] JSX Enterprises, JSX, http://jsx.org/, 2005-3-26.

[47] Kenneth B. Sall, XML Syntax and Parsing Concepts, http://www.awprofessional.com/articles/article.asp?p=27006&seqNum=1, 2005-3-26.

[48] W3C, QName, http://www.w3.org/TR/xmlschema-2/#QName, 2005-3-26.

[49] W3C, SOAP Specifications, http://www.w3.org/TR/soap/, 2005-3-26.

[50] SUN, Java API for XML-Based RPC, http://java.sun.com/xml/jaxrpc/,2005-3-26.

[51] SUN, Java DataBase Connectivity, http://java.sun.com/products/jdbc/, 2005-3-26.

[52] University of Aarhus, Petri nets tools, http://www.daimi.au.dk/PetriNets/tools/, 2005-3-26.

[53] University of Aarhus, CPN ML, http://www.daimi.au.dk/designCPN/sml/cpnml.html, 2005-3-26.

[54] OMG, CORBA, http://www.corba.org/, 2005-3-26.

[55] Apache, Apache AXIS, http://ws.apache.org/axis/, 2005-3-26.

[56] OsmoticWeb, WSDD, http://www.osmoticweb.com/axis-wsdd/, 2005-3-26.

[57] Michael B. Feldman, Software Construction and Data Structures with Ada 95,1996, Addison-Wesley Publishing Company.

[58] SUN, Using a Custom RMI Socket Factory, http://java.sun.com/j2se/1.4.2/docs/guide/rmi/socketfactory/, 2005-3-26.

[59] Jack Herrington, Code Generation in Action, 2003, Manning Publications.

Appendix A: MySQL database setup

The MySQL database setup is shown in the following steps:

1. Download softwares

   Our project uses MySQL for the storage. MySQL can be get from web: http://www.mysql.com. Also we need the JDBC driver to access and modify database, it can be get from the web: http://www.mysql.com/downloads/api-jdbc.html. All the Jar files should be included into the system classpath.

2. Create database and use it

   Open a dos window and input the command:

   create database nodea;

   This command will create nodea database in MySQL.

   Input another command to use the database-nodea

   use nodea;

3. Input the command to create user and password

   SET PASSWORD FOR 'root'@'localhost' = PASSWORD('root');

4. Create a table-trandata for the nodea database with a sql script:

   The example sql script for creating trandata table in nodea database is:

```
CREATE TABLE trandata
(
tranid VARCHAR(50) not null primary key,
status VARCHAR(20) ,
clientname VARCHAR(20) ,
clientmoney VARCHAR(20) ,
taskname VARCHAR(20) ,
taskcost VARCHAR(20) ,
tasktype VARCHAR(20) }
```

The tranid is primary key and not null; clientname and clientmoney are used for client object; taskname, taskcost, tasktype are used for task object. And status is used for saving transaction information.

The table-trandata can use created with the command in a dos window:

mysql nodea < create_client.sql

Similarly, we can create databases- nodeb, nodec for NodeB and NodeC sites and create tables for them.

Also if we want to see all the content of trandata table, we can use this command:

Select * from trandata;

The command will delete all content of trandata table:

Delete from trandata;

Appendix B: Setup of the Apache AXIS running environment

The Apache AXIS setup is shown in the following steps:

1. Download softwares

   We can get the free Apache AXIS distribution from the web: http://ws.apache.org/axis/, then unzip the file to a directory, for example c:\axis. Apache AXIS needs the XML parser to process XML messages. The Xerces 2 is recommended XML parser and can be downloaded from http://xml.apache.org/xerces2-j/index.html. After download, unzip the file to a directory: c:\xerces. Of course, we need SUN Java JDK for compile and running. All the jar files in these directories should be included in the system classpath.

2. Start three servers in different nodes:

   Open a DOS window, change the path to the directory which contains the Java codes and input the command:

   java org.apache.axis.transport.http.SimpleAxisServer -p 8080

   This command will run an AixsServer and listen the requests to the port : 8080.

   Similarly, we open other two DOS windows in two different machines, change the path to the directory which contains the Java codes and run two AxisServers on the same port.

3. Deploy the three services on three different AxisServers:

   Open one DOS window, change the path to the directory which contains the Java codes and .wsdd files and input the command:

   java org.apache.axis.client.AdminClient deploya.wsdd

   This command will deploy NodeA service in the AxisServer.

Similarly we open other two windows in other machines and input the commands respectively to setup NodeB, NodeC service:

java org.apache.axis.client.AdminClient deployb.wsdd

java org.apache.axis.client.AdminClient deployc.wsdd

4. Run the program:

   We can open an DOS command window in NodeA and input the command:

   java TestClient

   This command will send command to NodeA service and start the simulation.

Appendix C: Setup the Java RMI running environment and running instructions

1. Start Java RMI registry

   The Java RMI comes with Java JDK.

   For NodeA, open a window and input command to start rmi registry:

   rmiregistry

2. Start Java RMI servers

   Open another window and run the NodeA server:

   java NodeARMIServer

   Similarly, we can setup other two node RMI servers: NodeBRMIServer, NodeCRMIServer.

Appendix D: Setup J2EE running environment and running instructions

1. Download software

   In this project, I used JBoss(http://www.jboss.org) for J2EE container.

   The JBoss 3.x series are fully based on J2EE, and in one easy to install and easy to use package you will find all of the J2EE stacks and more.

   There are different versions of JBoss, one with just the JBoss core server that provides only an EJB server and another with Tomcat that includes a web container for servlets/JSP pages. We used JBoss with Tomcat bundle version.

   First make sure you have a recent JDK such as 1.3 or higher. The next step is obtaining JBoss from the JBoss website. You can find it at http://www.jboss.org/downloads.jsp , choose the jboss-3.0.4_tomcat-4.1.12.zip

   Then uzip the package to hard driver, you will find the directory:

   c:\ jboss-3.0.4_tomcat-4.1.12

   Rename it to:

   c:\jboss

   Setup the environment variable: Jboss_Home to c:\jboss

   Make sure you have included the jboss-j2ee.jar, jbossall-client.jar,jnp-client.jar, jnet.jar,jboss.jar, jnpserver.jar to your classpath.

   Put the project.jar to the :

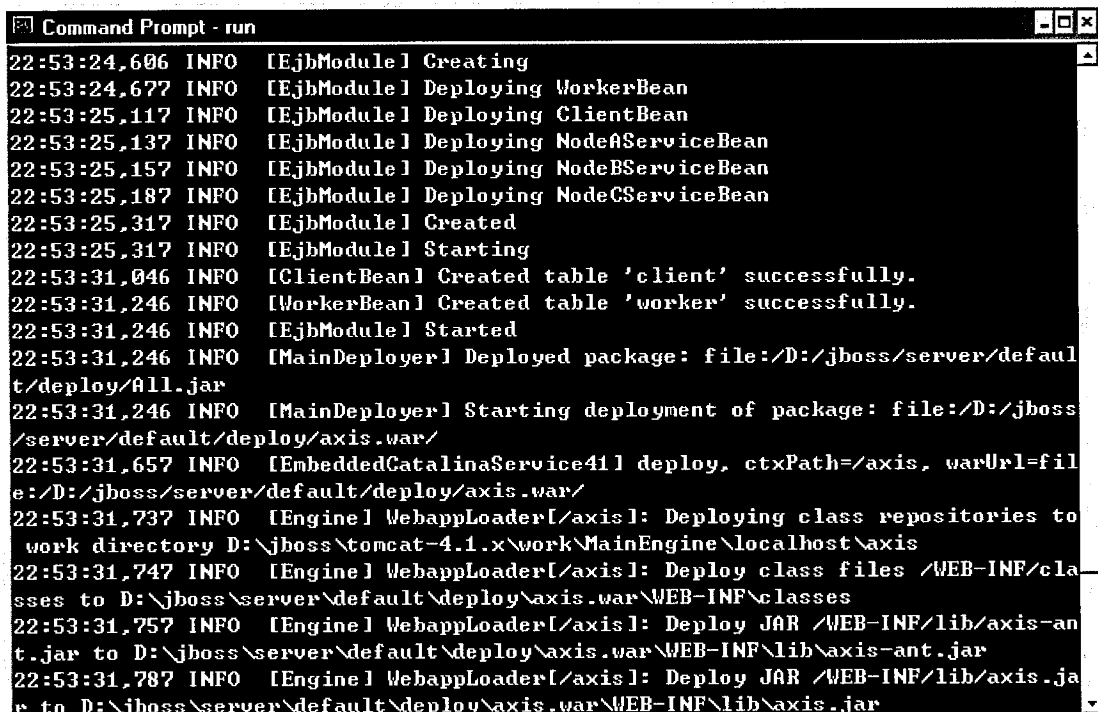
   c:\jboss\server\default\deploy

2. Start JBoss Servers

   Go to c:\jboss\bin directory.

Input: run

You can see a lot of output and the message of deployment in the following Figure:



From the screen shot, we can see that WorkerBean, ClientBean, TaskBean, NodeAService, NodeBService, NodeCService are deployed by JBoss, and JBoss also creates three Tables: client, worker for Database usage.

3. The deployment file: ejb-jar.xml

Another important issue in J2EE development is the deployment. In J2EE environment, we used three configuration files to setup session bean, entity bean and configure MySQL connection.

The ejb-jar.xml is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
```

```
<ejb-jar>
 <display-name>PetriNet Beans</display-name>
 <enterprise-beans>

  <entity>
   <description>Models a worker entitybean</description>
   <ejb-name>WorkerBean</ejb-name>
   <home>WorkerHome</home>
   <remote>Worker</remote>
   <ejb-class>WorkerBean</ejb-class>
   <persistence-type>Container</persistence-type>
   <prim-key-class>java.lang.String</prim-key-class>
   <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>WorkerBean</abstract-schema-name>
   <cmp-field><field-name>name</field-name></cmp-field>
   <cmp-field><field-name>money</field-name></cmp-field>
   <primkey-field>name</primkey-field>
  </entity>

 <entity>
   <description>Models a client entitybean</description>
   <ejb-name>ClientBean</ejb-name>
   <home>ClientHome</home>
   <remote>Client</remote>
   <ejb-class>ClientBean</ejb-class>
   <persistence-type>Container</persistence-type>
   <prim-key-class>java.lang.String</prim-key-class>
   <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>ClientBean</abstract-schema-name>
   <cmp-field><field-name>name</field-name></cmp-field>
   <cmp-field><field-name>money</field-name></cmp-field>
   <primkey-field>name</primkey-field>
  </entity>

  <session>
   <description>Models a NodeA service</description>
   <ejb-name>NodeAServiceBean</ejb-name>
   <home>NodeAServiceHome</home>
   <remote>NodeAService</remote>
   <ejb-class>NodeAServiceBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>
   <ejb-ref>
```

```
      <ejb-ref-name>ejb/Client</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <home>ClientHome</home>
      <remote>Client</remote>
      <ejb-link>ClientBean</ejb-link>
    </ejb-ref>
    <ejb-ref>
      <ejb-ref-name>ejb/Worker</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <home>WorkerHome</home>
      <remote>Worker</remote>
      <ejb-link>WorkerBean</ejb-link>
    </ejb-ref>
  </session>

  <session>
    <description>Models a NodeB service</description>
    <ejb-name>NodeBServiceBean</ejb-name>
    <home>NodeBServiceHome</home>
    <remote>NodeBService</remote>
    <ejb-class>NodeBServiceBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <ejb-ref>
      <ejb-ref-name>ejb/Client</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <home>ClientHome</home>
      <remote>Client</remote>
      <ejb-link>ClientBean</ejb-link>
    </ejb-ref>
    <ejb-ref>
      <ejb-ref-name>ejb/Worker</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <home>WorkerHome</home>
      <remote>Worker</remote>
      <ejb-link>WorkerBean</ejb-link>
    </ejb-ref>
  </session>

    <session>
    <description>Models a NodeC service</description>
    <ejb-name>NodeCServiceBean</ejb-name>
    <home>NodeCServiceHome</home>
    <remote>NodeCService</remote>
    <ejb-class>NodeCServiceBean</ejb-class>
    <session-type>Stateless</session-type>
```

```
<transaction-type>Container</transaction-type>
<ejb-ref>
  <ejb-ref-name>ejb/Client</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>ClientHome</home>
  <remote>Client</remote>
  <ejb-link>ClientBean</ejb-link>
</ejb-ref>
<ejb-ref>
  <ejb-ref-name>ejb/Worker</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>WorkerHome</home>
  <remote>Worker</remote>
  <ejb-link>WorkerBean</ejb-link>
</ejb-ref>
</session>
</enterprise-beans>
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>WorkerBean</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>ClientBean</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>
```

4. JBoss configuration file

There are two entity beans (ClientBean, WorkBean) and three session beans (NodeAServiceBean, NodeBServiceBean, NodeCServiceBean) in the file.

The jndi-name for each bean is defined in the jboss.xml file:

```
<?xml version="1.0" encoding="Cp1252"?>
<jboss>
  <secure>false</secure>
  <enterprise-beans>
    <entity>
```

```
        <ejb-name>WorkerBean</ejb-name>
        <jndi-name>test/Worker</jndi-name>
      </entity>
      <entity>
        <ejb-name>ClientBean</ejb-name>
        <jndi-name>test/Client</jndi-name>
      </entity>
    <session>
      <ejb-name>NodeAServiceBean</ejb-name>
      <jndi-name>test/NodeAService</jndi-name>
    </session>
    <session>
      <ejb-name>NodeBServiceBean</ejb-name>
      <jndi-name>test/NodeBService</jndi-name>
    </session>
    <session>
      <ejb-name>NodeCServiceBean</ejb-name>
      <jndi-name>test/NodeCService</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

5. JBoss and JDBC mapping file

   The jbosscmp-jdbc.xml file defines the relationship between EJB and MySQL.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jbosscmp-jdbc PUBLIC "-//JBoss//DTD JBOSSCMP-JDBC
3.0//EN"
"http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_0.dtd">
<jbosscmp-jdbc>
  <defaults>
   <datasource>java:/MySqlDS</datasource>
   <datasource-mapping>mySQL</datasource-mapping>
  </defaults>
  <enterprise-beans>
    <entity>
      <ejb-name>WorkerBean</ejb-name>
      <create-table>true</create-table>
      <remove-table>true</remove-table>
      <read-only>false</read-only>
      <table-name>worker</table-name>
      <cmp-field>
        <field-name>name</field-name>
        <column-name>name</column-name>
```

```
      </cmp-field>
      <cmp-field>
        <field-name>money</field-name>
        <column-name>money</column-name>
      </cmp-field>
    </entity>
     <entity>
      <ejb-name>ClientBean</ejb-name>
      <create-table>true</create-table>
      <remove-table>true</remove-table>
      <read-only>false</read-only>
      <table-name>client</table-name>
      <cmp-field>
        <field-name>name</field-name>
        <column-name>name</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>money</field-name>
        <column-name>money</column-name>
      </cmp-field>
    </entity>
    </enterprise-beans>
<dependent-value-classes>
</dependent-value-classes>
</jbosscmp-jdbc>
```

The MySQL connection password is set in the file – mysql-service.xml file.

Appendix E: Setup MySQL in JBOSS

1. Copy c:\jboss\docs\examples\jca\mysql-service.xml file to

   c:\jboss\server\default\deploy

2. Edit mysql-service.xml file to reflect your db connection info:

<config-property name="UserName" type="java.lang.String">root</config-property>

<config-property name="Password" type="java.lang.String">root</config-property>

3. Copy the library containing the DriverClass to c:\jboss\server\default\lib

4. Edit c:\jboss\server\default\conf\standardjbosscmp-jdbc.xml:

   Add an entry for your DS

   <datasource>java:/MySqlDS</datasource>

   <datasource-mapping>mySQL</datasource-mapping>

Appendix F: SOAP call example

During the first phase,NodeA will send a SOAP call to NodeB and ask NodeB ready for commit or not:

```
service = new Service();
call = (Call) service.createCall(); // create a new call
call.setTargetEndpointAddress(new
java.net.URL("http://localhost:8081/axis/servlet/AxisServlet") ); // set target address
call.setUseSOAPAction(true); // use SOAP Action
call.setSOAPActionURI("http://www.simpleprotocol.org/NodeB");
call.setMaintainSession(true); //establish session scope
QName p11 = new QName("http://www.simpleprotocol.org/NodeB",
              "Task"); // Set the mapping for bean-serializer
Class cls = Task.class;
call.registerTypeMapping(cls, p11, BeanSerializerFactory.class,
              BeanDeserializerFactory.class);
QName p22 = new QName("http://www.simpleprotocol.org/NodeB",
              "Client");
cls = Client.class;
call.registerTypeMapping(cls, p22, BeanSerializerFactory.class,
BeanDeserializerFactory.class);
QName p33 = new QName("http://www.simpleprotocol.org/NodeC",
              "Client");
cls = Client.class;
call.registerTypeMapping(cls, p33, BeanSerializerFactory.class,
BeanDeserializerFactory.class);
call.addParameter("source", XMLType.SOAP_VECTOR, ParameterMode.IN);
call.setOperationName( new QName("http://www.simpleprotocol.org/NodeB",
              "RequestFromA" )); // call the nodeb method
call.setReturnType( org.apache.axis.encoding.XMLType.XSD_STRING );
call.setTimeout(new Integer(10000));        // Set the timeout of the call
ttt = (String)call.invoke( new Object[] {vect});
```

Let us look how it works:

- initiate a new SOAP call
- setup the target node address
- use SOAP action in the call
- set up the Bean-Serializer mapping for classes
- add parameter which is a String to the SOAP call

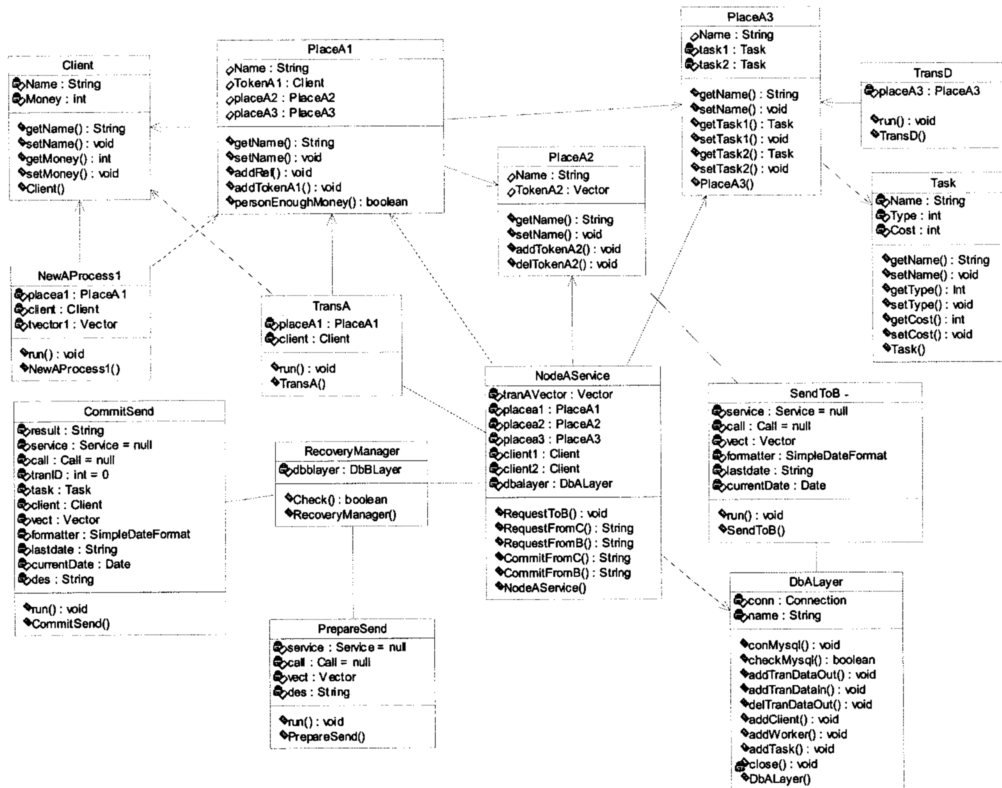
- set operation name
- set the return type to String
- set the timeout of the call
- make the call

After the NodeB gets the call, it will return the ready or not to NodeA in the same call.

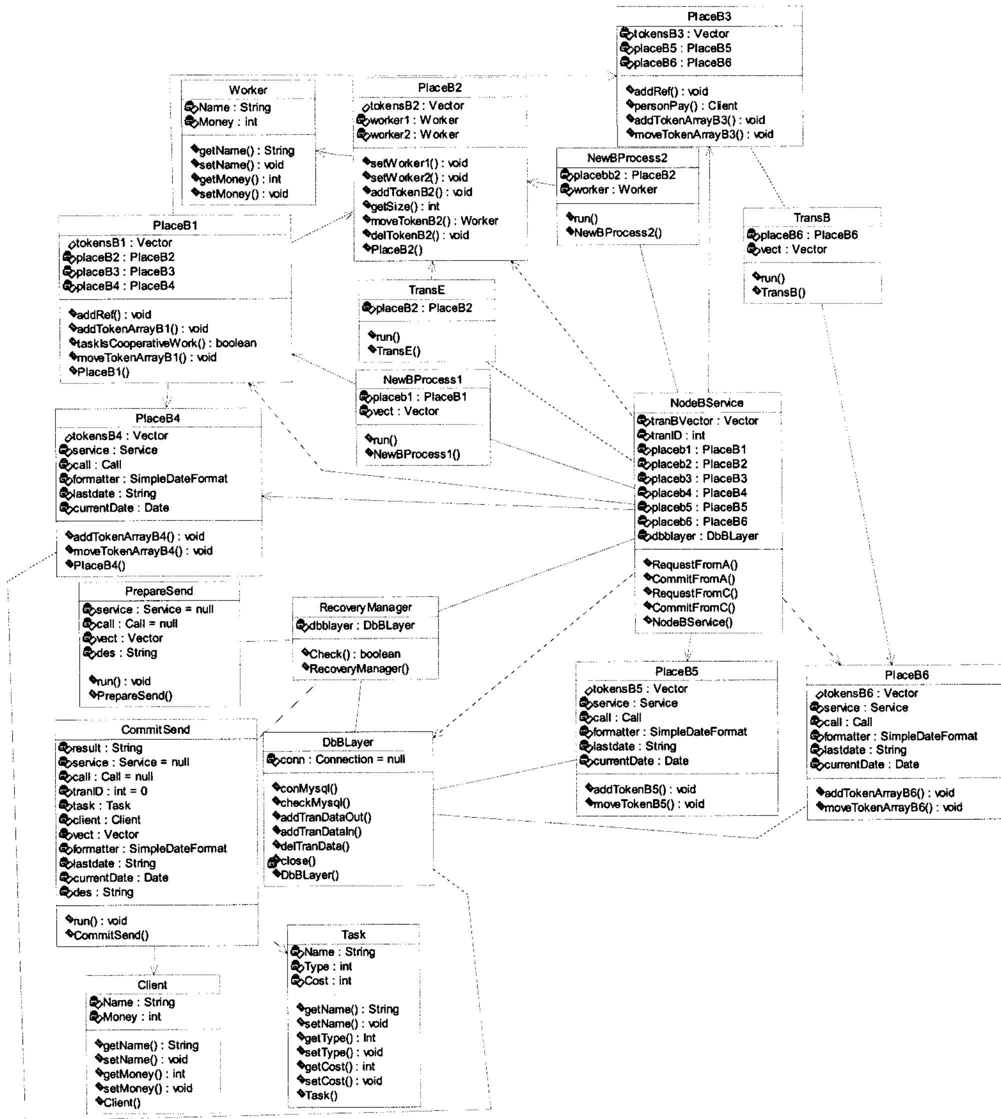
During the phase two, NodeA will send commit SOAP call to NodeB:

```
service = new Service();
call = (Call) service.createCall();
call.setTargetEndpointAddress(new
 java.net.URL("http://localhost:8081/axis/servlet/AxisServlet") );//Set target node
call.setUseSOAPAction(true);     // use SOAP Action
call.setSOAPActionURI("http://www.simpleprotocol.org/NodeB");
call.setMaintainSession(true); //establish session scope
call.addParameter( "op1", XMLType.XSD_STRING, ParameterMode.IN );
call.setOperationName( new QName("http://www.simpleprotocol.org/NodeB",
                                                        "CommitFromA" ));
call.setReturnType( org.apache.axis.encoding.XMLType.XSD_STRING );
call.setTimeout(new Integer(10000));          // Set the timeout of the call
ttt = (String)call.invoke( new Object[] {i1}); // make the SOAP call
```

Appendix G: Java class diagrams



Java RMI approach: Class diagram for NodeA



Java RMI approach: Class diagram for NodeB

Client
Name : String
Money : int
getName() : String
setName() : void
getMoney() : int
setMoney() : void
Client()

PlaceA1
Name : String
TokenA1 : Client
placeA2 : PlaceA2
placeA3 : PlaceA3
getName() : String
setName() : void
addRef() : void
addTokenA1() : void
personEnoughMoney() : boolean

PlaceA3
Name : String
task1 : Task
task2 : Task
getName() : String
setName() : void
getTask1() : Task
setTask1() : void
getTask2() : Task
setTask2() : void
PlaceA3()

TransD
placeA3 : PlaceA3
run() : void
TransD()

PlaceA2
Name : String
TokenA2 : Vector
getName() : String
setName() : void
addTokenA2() : void
delTokenA2() : void

Task
Name : String
Type : int
Cost : int
getName() : String
setName() : void
getType() : int
setType() : void
getCost() : int
setCost() : void
Task()

NewAProcess1
placea1 : PlaceA1
client : Client
tvector1 : Vector
run() : void
NewAProcess1()

TransA
placeA1 : PlaceA1
client : Client
run() : void
TransA()

NodeARMIServer
placea1 : PlaceA1
placea2 : PlaceA2
placea3 : PlaceA3
client1 : Client
client2 : Client
NodeARMIServer()
main() : void

SendToB
service : Service = null
call : Call = null
vect : Vector
formatter : SimpleDateFormat
lastdate : String
currentDate : Date
run() : void
SendToB()

CommitSend
result : String
service : Service = null
call : Call = null
tranID : int = 0
task : Task
client : Client
vect : Vector
formatter : SimpleDateFormat
lastdate : String
currentDate : Date
des : String
run() : void
CommitSend()

RecoveryManager
dbblayer : DbBLayer
Check() : boolean
RecoveryManager()

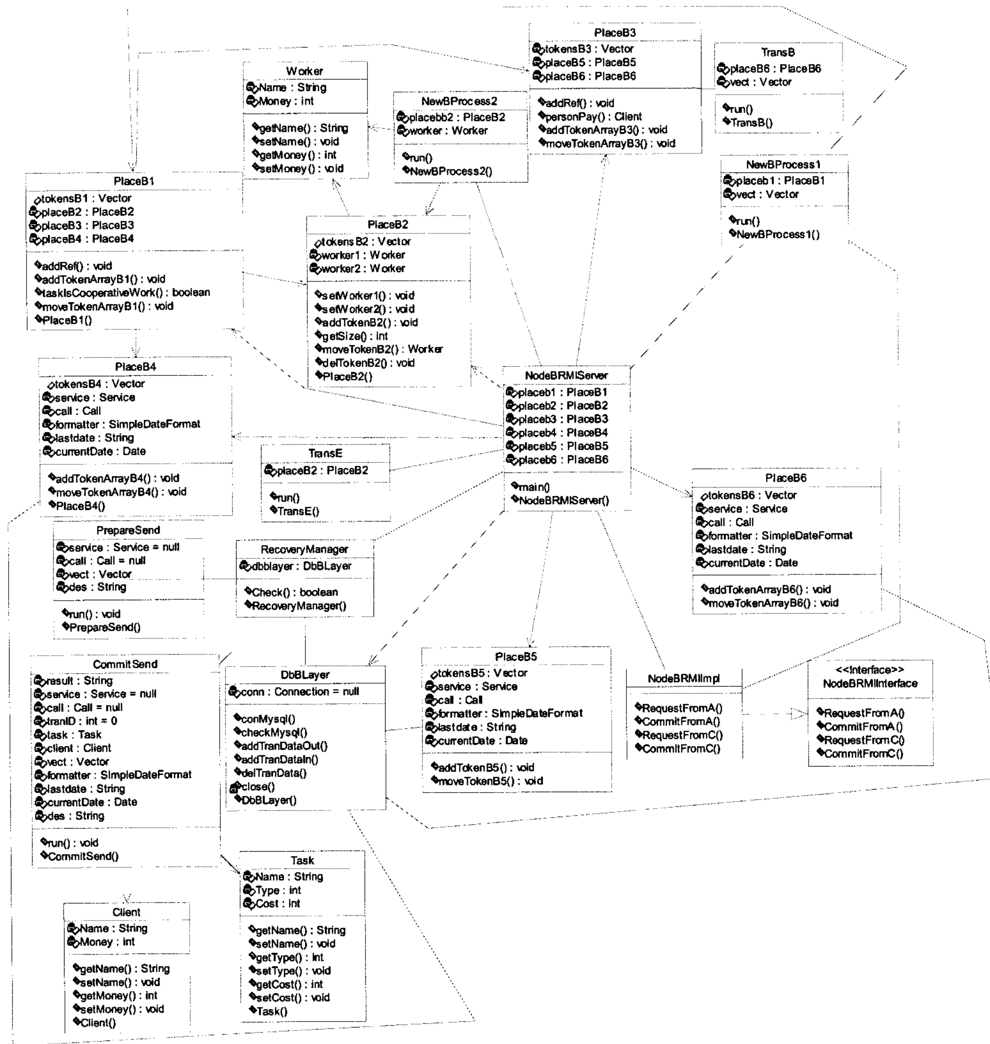
<<Interface>>
NodeARMIInterface
RequestToB()
RequestFromC()
RequestFromB()
CommitFromC()
CommitFromB()

PrepareSend
service : Service = null
call : Call = null
vect : Vector
des : String
run() : void
PrepareSend()

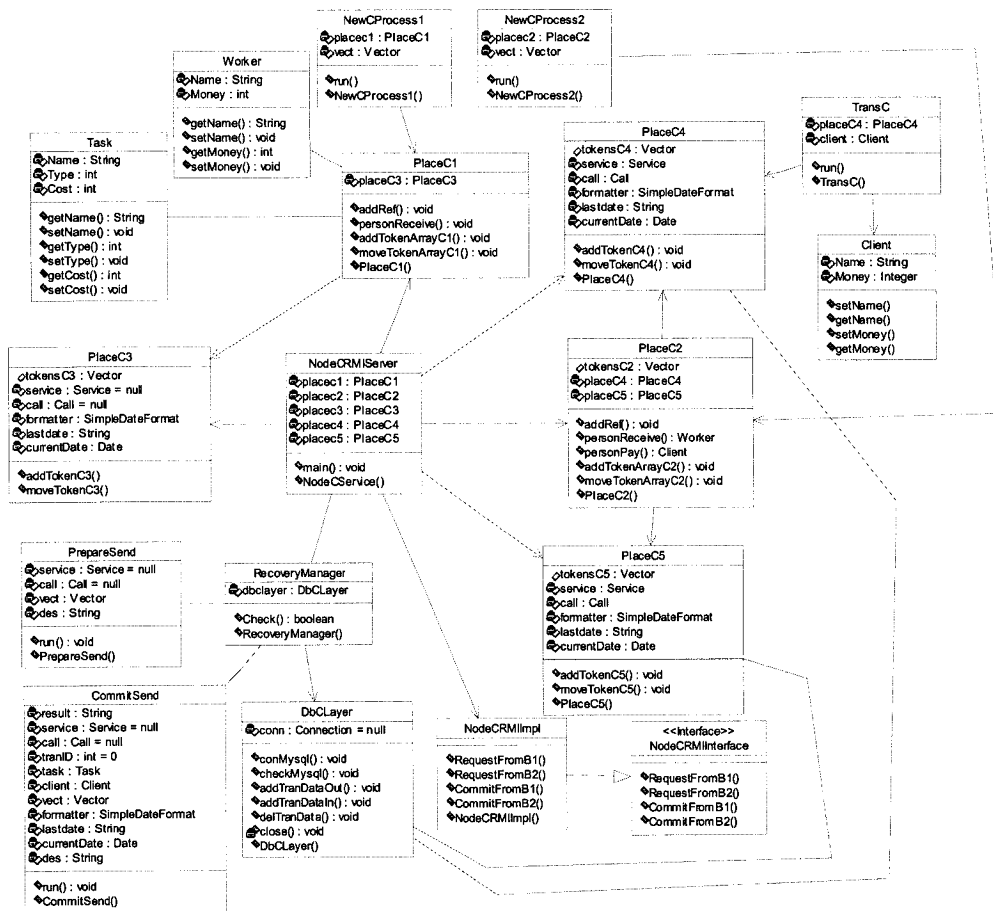
DbALayer
conn : Connection
name : String
conMysql() : void
checkMysql() : boolean
addTranDataOut() : void
addTranDataIn() : void
delTranDataOut() : void
addClient() : void
addWorker() : void
addTask() : void
close() : void
DbALayer()

NodeARMIImpl
RequestToB()
RequestFromC()
RequestFromB()
CommitFromC()
CommitFromB()
NodeARMIImpl()

Java RMI approach: Class diagram for NodeC

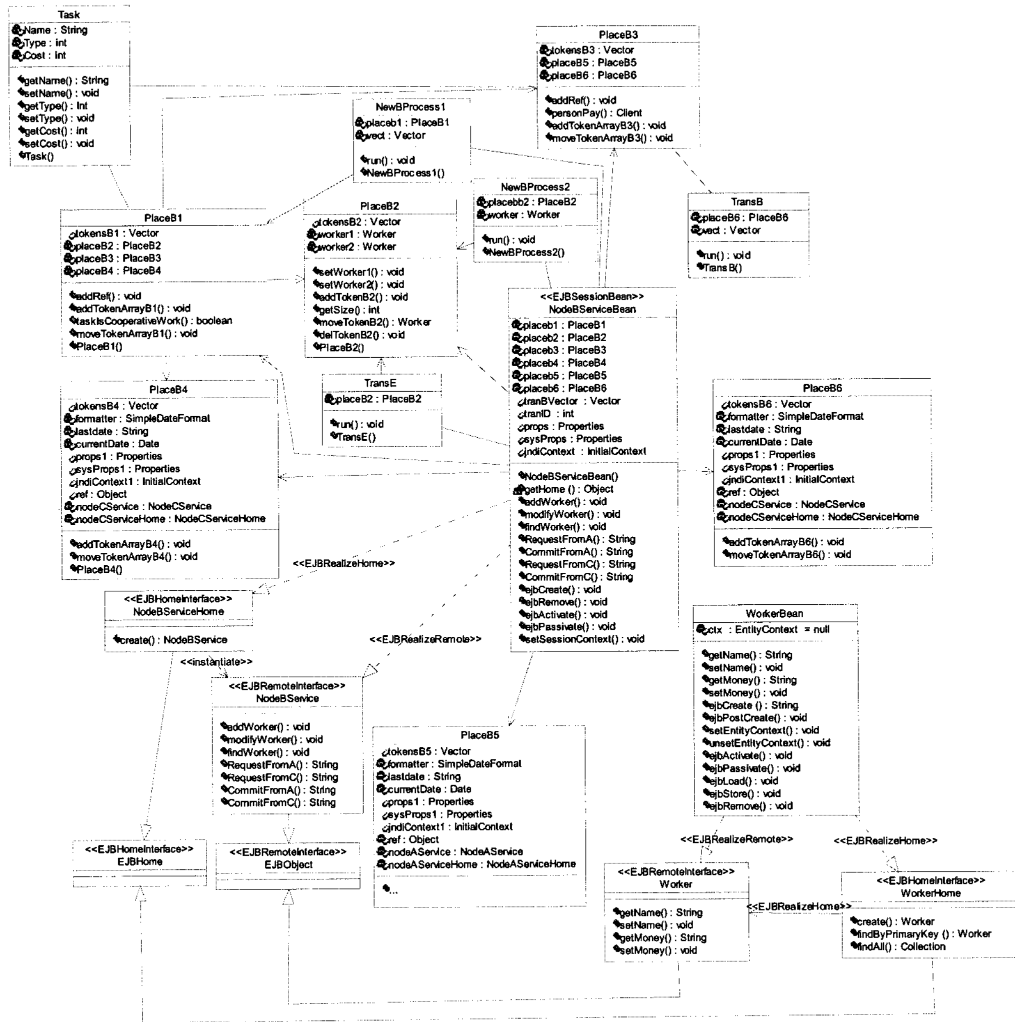


SOAP approach: Class diagram for NodeA

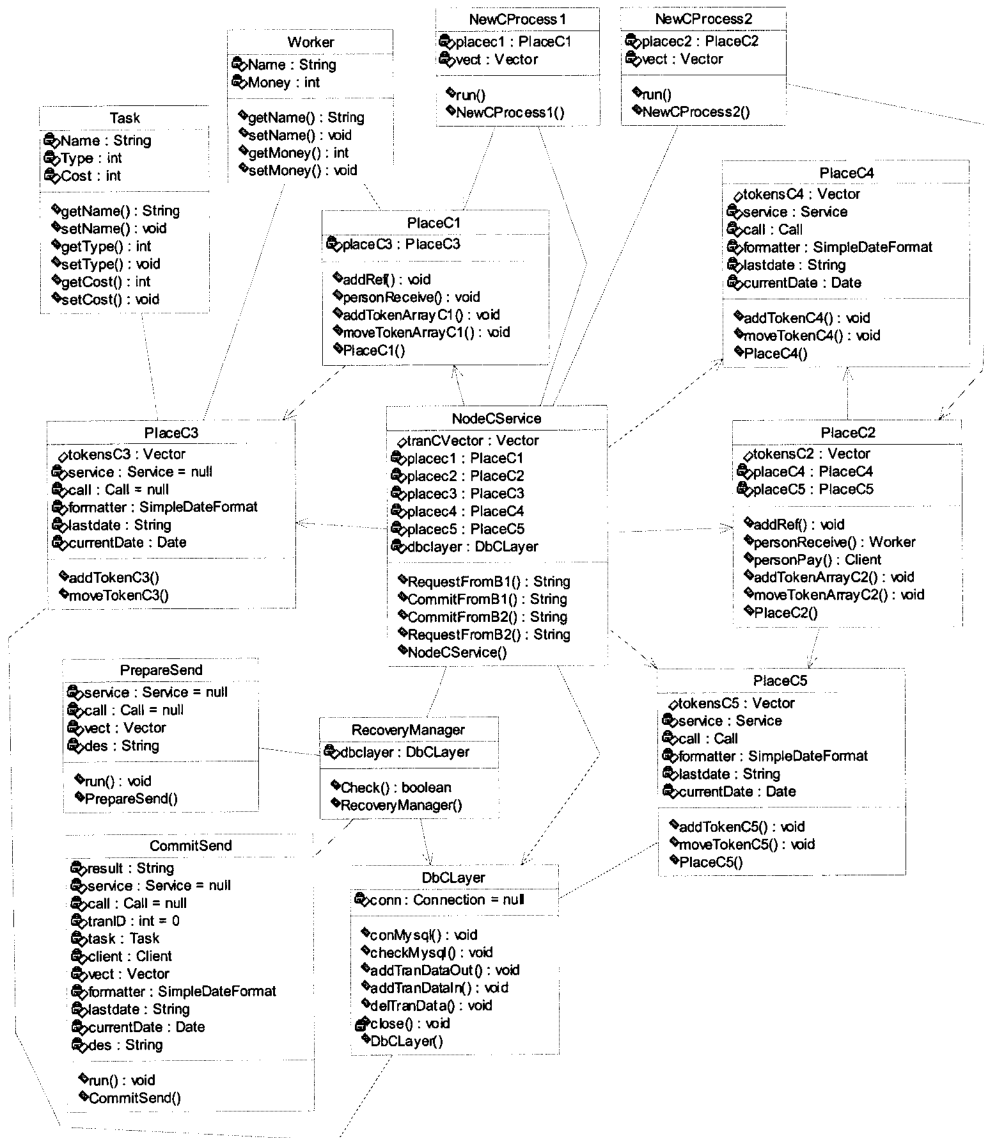


SOAP approach: Class diagram for NodeB

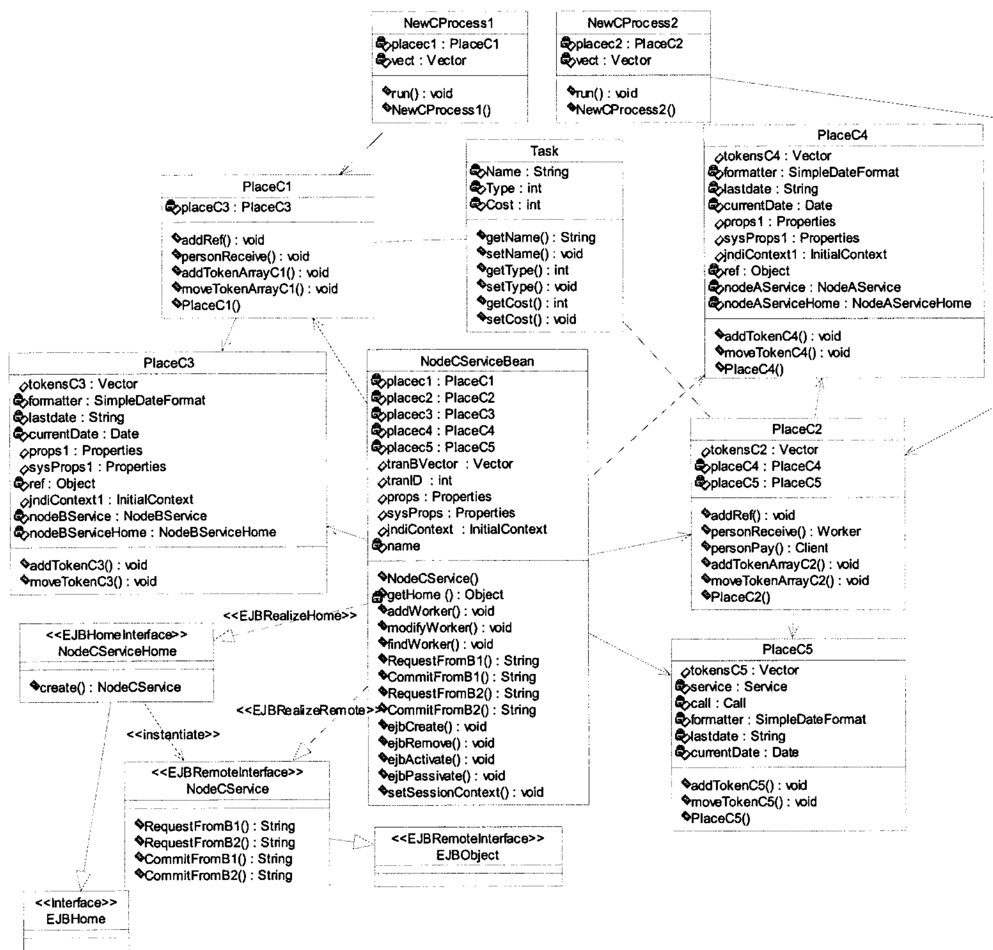
SOAP approach: Class diagram for NodeC



J2EE approach: Class diagram for NodeA



J2EE approach: Class diagram for NodeB



J2EE approach: Class diagram for NodeC

Appendix H: Message length comparison

The Client object with the same values is sent from a client to a server. The values of the Client parameters are: Name with the value of "David", Money with the value of 100.

- XML message length

  We can use the code below on the server side and get the XML message length:

```
SOAPMessageContext soapMsgContext= (SOAPMessageContext)
(serviceContext.getMessageContext());
SOAPMessage soapm = soapMsgContext.getMessage();
ByteArrayOutputStream bb = new ByteArrayOutputStream() ;
soapm.writeTo(bb); // Write SOAPMessage to ByteArrayOutputStream
System.out.println(bb.size()); // Get message length
```

  The idea is that we get the SOAPMessage from the SOAPMessageContext, write SOAPMessage to ByteArrayOutputStream then get the XML message length.

  The XML message length is 930 bytes for the sending the Client object.

- Java RMI message length

  Java RMI uses and encapsulates sockets as part of its communication protocol. In order to get the Java RMI message length, we may write our custom RMI socket factory (see [58]):

  1. Implement a custom ServerSocket and Socket.
  2. Implement a custom RMIClientSocketFactory.
  3. Implement a custom RMIServerSocketFactory.
  4. Implement an InputStream for the custom socket.
  5. Implement an OutputStream for the custom socket.

  Then we can calculate how many bytes the OutputStream of the client socket writes. The Java RMI message length is 590 bytes for the sending the Client object.