Developing component-based Web transaction systems with Servlet/JSP and CORBA.

Nan. Zhang
University of Windsor

DEVELOPING COMPONENT-BASED WEB TRANSACTION SYSTEMS WITH SEERVLET/JSP AND COBRA

by

Nan Zhang

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2000

0-612-62309-2

Abstract

Web has evolved from a network of basically static information display to a mechanism for interactive Web application. Currently, web creates for enhancing business processes, reducing costs and increasing profits. With component-based computing and MVC design pattern, web application could be developed easily, less expensively, deployed flexibly across platform, adapted to new technology quickly.

Rapid delivery of business-critical information over the Internet and Corporate intranets requires a transaction-processing solution that integrates the functionality of multiple objects, provides these objects with access to multiple data sources, and ensures data integrity, scalability, and security across every business transaction. Moreover, certain runtime services should developed to support Web applications. The ability to easily locate application components, access and execute them securely, and ensure their availability are critical factors to deploying Web applications in Internet, intranet, and extranet environments.

To meet increasing need, we develop a viable approach for web transaction systems based on the component-based system development in this thesis work. This approach adopts the distributed, component-based development principle and is based on MVC design pattern. This approach offers the flexibility, scalability, and reliability necessary to support the constant evolution of business processes in the e-business world. And a prototype, which based on our infrastructure design, is also developed to validate the feasibility and effectiveness of the proposed approach.

Keywords: Component-based Web Transaction Systems (CWTS), Component-based System Development (CBD), Integrated Transaction Service (ITS), Object Transaction Service (OTS), Model/View/Controller (MVC) design pattern, Web application, Java Servlet, Java Server Page (JSP), JavaBeans component, Unified Modeling Language (UML)

To My Parents and My Family

Acknowledgements

I would like to acknowledge foremost the constructive guidance of my supervisor, Dr. Tjandra. Without his patience, knowledge and encouragement, this thesis could not have been completed.

I wish to thank the members of my committee, Dr. Kaloni, external reader from Department of Mathematics and Statistics, Dr. Bandyopadhyay, internal reader, and Dr. Aggarwal, chair of the committee, from Department of Computer Science, for their valuable comments, suggestions and clarification of a few points in the thesis.

Also, I would like to thank my parents. In particular, I am greatly indebted to my father for his love, encouragement, and support in my life. Without his contribution for taking care of my daughter during the period of my graduate studies, it would be impossible for me to have completed my graduate study program.

Finally, I am grateful to Dr. Zongchun Qiao, my husband, and Katherine Qiao, our daughter, whose support and understanding was also vital to the completion of my thesis.

List of Figures

Figure 3.1	OMA Reference Model	25
Figure 3.2	CORBA Architecture with its Essential Elements	28
Figure 4.1	Simple Distributed Transaction	33
Figure 4.2	Nested Distributed Transaction	34
Figure 4.3	Object Transaction Service Entities	37
Figure 5.1	the homepage of Java Bookstore	47
Figure 5.2	System Architecture	49
Figure 5.3	Functional Modules for the Prototype	52
Figure 5.4	Shopping Scenario Use Case Diagram	61
Figure 5.5	Transaction Scenario Class Diagram	62
Figure 5.6	Transaction Scenario Interaction Diagram	63

Table of Contents

Abstract iii

Dedication iv

Acknowledgements v

List of Figures vi

1 Introduction 1

1.1 Motivations and Objectives 1

1.2 Thesis Structure 5

2 Background: Distributed System, Component-Based System Development, Multitier Architecture, and Web Application 6

2.1 Distributed Systems 6

2.2 Component-Based System Development 10

2.2.1 Component-Based System Development 10

2.2.2 Component Models 12

2.2.3 JavaBeans Component Model 13

2.3 Multitier Architecture 13

2.4 Web Application 16

2.4.1 Web Application Architecture 16

2.4.2 Web Client Models 17

2.4.3 Traditional Approaches 18

2.4.4 Servlet/JSP Technologies 19

3 OMA and CORBA 23

3.1 Object Management Architecture 23

3.2 OMA Reference Model 23

3.3 Overview of CORBA 25

3.3.1 Essential Components of CORBA 26
3.3.2 CORBA Naming Service 28

4 Distributed Transaction Management 31
4.1 Concept of Transaction 31
4.2 Distributed Transactions 33
4.2.1 Classification of Distributed Transaction 33
4.2.2 Distributed Transaction Protocols 34
4.3 Distributed Transaction Management Standards 35
4.3.1 Microsoft Transaction Server 35
4.3.2 X/Open Distributed Transaction Processing (DTP) Model 36
4.3.3 OMG Object Transaction Service 36
4.3.4 Java Transaction Service 38
4.3.5 VisiBroker Integrated Transaction Service (ITS) 39
4.3.5.1 VisiBroker Integrated Transaction Service Specification: IDL Interfaces 39
4.3.5.2 Models for Transactions 40
4.3.5.2.1 Model for Basic Transactions 40
4.3.5.2.2 Model for Transaction with Resource Objects 41
4.3.5.3 The Advantages of VisiBroker's ITS 43

5 System Design and Implementation 45
5.1 Requirements Analysis 45
5.2 System Design Goals and Considerations 46
5.3 System Architecture 47
5.3.1 A Typical Web Site 47
5.3.2 System Model – Distributed Multitier Component-Based Model 48
5.3.3 The Architecture of the Prototype 50
5.3.4 MVC Architecture 52
5.3.4.1 MVC Design Pattern 53
5.3.4.2 The View 54

5.3.4.3 The Model 55

5.3.4.4 The Controller 56

5.4 System Functionality 57

5.4.1 Shopping Scenario 57

5.4.2 Transaction Scenario 58

5.5 System Modeling with UML 60

5.5.1 Shopping Scenario Modeling 61

5.5.2 Transaction Scenario Modeling 61

5.6 The Contracts among Transaction Components – IDL Interface 64

5.6.1 The IDL for ITS Transaction Service Standard 64

5.6.2 The IDL for ITS Transaction Service Extensions 68

5.6.3 The IDL for the Prototype 69

5.7 Detailed Transaction Implementation of Prototype 70

5.8 Development Environment 73

5.8.1 Platform 73

5.8.2 Development Tools 74

5.9 The Operation of the System 75

6 Conclusions and Future Works 78

6.1 Conclusions 78

6.2 Future Works 79

Appendix A: List of Packages and Files in the System 81

Bibliography 83

Vita Auctoris 87

1 Introduction

In this thesis, we are primarily concerned with the discussion of the development of ensuring integrated transaction for Web applications in the distributed environment. The infrastructure design presented here is based on the principles of distributed, multitier architecture, and component-based approach, and the use of open standards and multi-platform technologies, and MVC design pattern. As a prototype, a sample on-line bookstore is developed with the help of UML modeling, to demonstrate the feasibility and effectiveness of our approach.

In the remainder of this introductory chapter, we first address the motivations and objectives of this thesis in *Section 1.1*. The overall organizations of this thesis are then outlined in *Section 1.2*.

1.1 Motivations and Objectives

Few technologies have revolutionized business more than the advent of the Internet. In its short history, the Internet began as a giant unidirectional medium, or a giant URL-based file server, for publishing and broadcasting static electronic documents. Since the mid of 1990's, companies all over the world have been quickly realizing that the Internet's true value is not in people's ability to browser the Web, but rather, in the new opportunities it creates for enhancing business processes, reducing costs and increasing profits. As a result, the Web has evolved from a network of basically static information display to a mechanism for interactive Web applications.

On the other hand, facing growing pressure to delivery broad-reaching application functionality quickly and cost-effectively, the software industry are turning to distributed object computing as the adaptive software architecture on which to build applications that meet these challenging business requirements. With object-oriented computing, organizations can develop applications faster and less expensively by reusing software components; adapt to new technology quickly while integrating existing system; deploy applications flexibly across platform; and simplify maintenance by isolating volatile code in frequently changed objects. The explosion of the Internet has fueled this shift to distributed objects. Internet business opportunities require a new breed of information technology solutions that are suited to this new, rapidly expanding Web computing world.

Facing increasingly sophisticated web applications require changes in development technology. What is need is an industry-wide solution for creating pages with dynamic-generated content. This solution should address the limitations of current alternatives by:

- Working on any web or application server
- Separating the application logic from the appearance of the page
- Allowing fast development and testing
- Simplifying the process of developing interactive web applications

The Servlet/JSP technology was designed to fit the above industry need. Java Servlet/JSP is more efficient, easier to use, more powerful, more portable, safer, and cheaper than traditional CGI and many alternative CGI-like technologies. Microsoft's Active Server Pages™ (ASP) technology makes it easier to create dynamic content on a web page, but only works with Microsoft IIS or Personal Web Server. Another early solution to this problem was the CGI-BIN technology. With traditional CGI, a new process is started for each HTTP request. The overhead of starting the process can dominate the execution time. With Servlet/JSP, the Java Virtual Machine stays running and handles each request using a lightweight Java thread, not a heavyweight operating system process. This is the major motivation why we adopt the Java Servlet/JSP technology to build our component-based web transaction systems infrastructure.

As a growing number of web applications are deployed over the Internet, the need arises to coordinate application objects into Internet transactions. Web applications typically require flexible access to multiple data sources, such as inventory, customer, or shipping data, while maintaining data integrity across these data sources to ensure a consistent state. Such applications create the need for a software solution that coordinates their activities, ensuring consistence, isolation, and durability [Subr99]. Traditional approach to manage atomic transaction is TP (Transaction Processing) Monitor technology. The X/Open DTP model is well established in the industry for 25 years [TPMT97]. Most of the commercial databases such as Oracle, Sybase implement XA interface of TP technology to perform business logic computations and database updates. But the X/Open DTP model has not efficiently addressed the interoperability issues across different TP domain and its implementation code is usually written in a lower-level language (such as COBOL) [TPMT97]. Facing with today's complexity of business need in distributed environment, integrate more powerful distributed transaction processing service is important. To achieve this goal, Web applications require an object-based transaction service that enables the following

- Coordination of multiple component into a single transaction.
- Multiple components to access a single data source or multiple data sources.
- Data integrity across multiple data sources.
- Security of business over the Internet or corporate Intranets.

In order to remain competitive, many companies have the need to combine the flexibility and reuse of object technology with the application services that now define customer service: performance, reliability, transaction integrity, and security. Such that, one of our thesis objective is combining standards-based solution for this problem, which integrate OMG Object Transaction Service technology into our web system design and provides more enhancement over traditional transaction management.

Moreover, Web application is not simply defined as an e-commerce transaction. It is about using technology to redefine old business models in order to maximize customer value [WAPM99]. With the relentless evolution of Web technology, non-Web application models and organizational structures will face increasing pressure. In order to compete, some might even survive, in this new economic era; a company must be able to react quickly to challenges by constantly innovating their processes to stay in step with new technologies and ahead of competition. To be successful in the e-business world, companies thus need to create an e-business infrastructure, which is a set of tools that enables e-business. The infrastructure thus must be designed to meet the following key criteria:

- Flexibility: for rapidly evolving Web application models through the addition of new application functionality and the integration of systems and applications with customers, business partners and suppliers.
- Scalability: for accommodating unpredictable fluctuations in custom demand and user workload.
- Reliability: for ensuring secure, continuous operation and availability of the Web applications to end-users.

Unfortunately, although a lot of Web applications are available today, many of them may not met the infrastructure design criteria and/or supported the transactions as we have described above. Driven by the motivations to address these two needs, in this thesis, we are primarily concerned with the discussion for developing component-based web transaction systems in the distributed environment. The proposed approach is based on the principles of distributed, multitier architecture, component-based system development [Brown98a, 98b] with the use of open standards, multi-platform technologies, and MVC design pattern [GHJV95] as well. As a prototype of the framework discussed here, a sample on-line bookstore is developed, with the help of UML modeling, to demonstrate the feasibility and effectiveness of our approach. The major benefits of the proposed approach may include the following:

- Simplify application development and deployment.
- Support heterogeneous client and server platforms.
- Deliver a secure, scalable, reliable, and manageable environment.

Accordingly, the primary objectives of this thesis may thus be summarized as follows:

- To propose an approach for the development of component-based web transaction systems in the distributed environment. This approach offers the flexibility, scalability, and reliability necessary to support the constant evolution of business processes in the e-business world.

- To develop a prototype based on our infrastructure design to validate the feasibility and effectiveness of the proposed approach. This prototype is expected to be adapted to a wide range of e-commerce application with only some modifications and extensions.

It is worth emphasizing that these objectives achieved in this thesis lies in the fact that our framework is designed based on the principles of distributed, multitier architecture, component–based development approach with the use of open standards, industry standard component models (CORBA [Siegel96] and JavaBeans [Sun97j]), multi-platform technologies (JSP/Servlet technologies [SS99], and Java programming language [JLS97]), together with the help of MVC design pattern [GHJV95] and UML modeling [BRJ99]. These principles and technologies will be introduced and discussed in the rest of the chapters.

1.2 Thesis Structures

We conclude this introductory chapter with an outline of the remainder chapters as follows. In Chapter 2, we provide an overview of the background information to this thesis. In particular, the characteristics and advantages of distributed computing, multitier architecture, and component-based development approach as well, are discussed. They are the principles and guidelines for the system architecture design of our Web application. The Web application with its architecture, Web client models, and traditional approaches are then introduced. JavaBeans, one of the component model adopted in this thesis, and JSP/Servlet technologies are also briefly introduced in this chapter since they will be chosen for our Web client tier design. An overview of CORBA technology, another component model adopted for our infrastructure of Web application, is provided in Chapter 3. After an introduction of the general concepts and characteristics of distribute transaction management, in Chapter 4, we present an overview of available transaction services with focus on OTS, one of implementation of CORBA Transaction Service, employed for the transaction management in our infrastructure. The system design and implementation of our prototype, together with MVC design pattern and UML modeling, are detailed in Chapter 5. The conclusion and future works are summarized in Chapter 6.

2 Background: Distributed System, Component-Based System Development, Multitier Architecture, and Web Application

In this chapter, we introduce some background information related to this thesis. We first provide an introduction to the characteristics and advantages of distributed system, component-based system development approach, and multitier architecture. These are the principles and guidelines for the system architecture design of our Web application. The Web application's architecture, Web client models, and traditional approaches with its disadvantages are then introduced and discussed. JavaBeans and JSP/Servlet technologies are also briefly introduced in this chapter since these three technologies, following the MVC architecture, will be chosen for our Web client tier design, which is detailed in Chapter 5.

2.1 Distributed Systems

A *distributed system* consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software [Cou96]. Distributed system software enables computers to coordinate their activities and to share the resources of the system-hardware, software and data. Users of a well-designed distributed system should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations.

The development of distributed systems followed the emergence of high-speed local area computer networks at the beginning of the 1970s. More recently, the availability of high-performance personal computers, workstation and server computers has resulted in a major shift towards distributed systems and away from centralized and multi-user computers. This trend has been accelerated by the development of distributed system software, designed to support the development of distributed application. For an in-depth introduction of knowledge of distributed system, we refer to the excellent book by Goulouris et. al. [Cou96]. In the following, however, for the purpose of completeness, we provide a summary of six key characteristics that are primarily for the usefulness of distributed system.

- **Resource sharing:** Resources (that may extend the range from hardware components such as disks and printers to software-defined entities such as files, windows, database and other data objects) in a distributed system are physically encapsulated with one of the computers and can only be accessed from other computers by communication. For effective sharing each resource must be managed by a program, called resource manager, that offers a communication interface enabling the resource to be accessed, manipulated and updated reliably and consistently.
- **Openness:** The openness of a computer system is the characteristic that determines whether the system can be extended in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added without disruption to or duplication of existing service. Open distributed systems are based on the provision of a uniform inter-process communication mechanism and published interfaces for access to shard resources. Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors.
- **Concurrency:** Concurrency and parallel execution arise naturally in distributed systems from the separate activities of users, the independence of resources and the location of server processes in separate computers. The separation of these activity enables processing to proceed in parallel in separate computers, and accordingly results in the high performance benefit of distributed systems. However, concurrent accesses and updates to shared resources must be synchronized to ensure they do not conflict with each other, and to keep shared resource in a consistent state.
- **Scalability:** Scalability is defined as that the system and application software should not change when the scale of the system increases. Scalability has been a dominant concern in distributed system design during the last decade, and its importance continues due to the size and complexity of computer networks growing. Current available techniques including replicated data, caching and deployment of multiple servers, have been successfully in coping with large-scale applications.

- **Fault tolerance:** Computer systems sometimes may fail due to the fault occurring in the hardware or software. As a result, this failure may lead to the running program to produce incorrect results, or terminates without the completion of the intended computation. Fault tolerance can be addressed more efficiently in distributed systems than in more centralized system architectures. Hardware redundancy (the use of redundant components) and software recovery (the design of programs to recover from faults) are two approaches widely used for the design of fault-tolerant computer system. Hardware redundancy can be exploited to ensure that essential tasks are re-allocated to another computer when one fails. Software recovery involves the design of software so hat the state of permanent can be recovered or "rolled back" when a fault is detected.
- **Transparency:** Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. The implications of transparency are a major influence on the design of the software system. Based on the functionality, transparencies may further be classified into eight forms as, transparency of access, location, concurrency, replication, failure, migration, performance, and scaling. These provide a useful summary of the motivation and goals for distributed systems. It is worth noting that among them, access transparency-enables local and remote information objects to be accessed using identical operations, and local transparency-enables information objects to be accessed without knowledge of their location are two most important transparencies. They are sometimes referred to together as network transparency.

It is worth emphasizing that a distributed system may offer substantial benefits summarized by the above six characteristics, to their users. Nevertheless, these benefits cannot be achieved automatically without a careful design of system components. Distributed systems have become the norm for the organization of computing facilities. They can be used to implement general-purpose interactive computing systems, and support a wide range of commercial and industrial application of computers. They are increasingly being used as the basis for new applications where communication is a basic requirement.

2.2 Component-Based System Development

In this section, we first present an overview of the concept and advantages of component-based system development approach. The concepts of component and component models are then discussed. Finally, JavaBeans, one of the popular component models, is briefly introduced.

2.2.1 Component-Based Development

Recently, there has been renewed interest in the notion of software development through the planned integration of pre-existing software components. This is most often called component-based development (CBD) [Barn98, Brown96, Brown98a]. CBD is an approach to application development that moves away from tradition, custom development to assembly from pre-built, pre-tested software components, which inter-operate with each other in a distributed, networked environment [Stuart98].

Conceptually, CBD unifies concepts from a number of software domains, such as object-oriented programming, software architecture, and distributed computing. CBD emphasizes the outsourcing of pieces of the application system, and focuses on the controlled assembly of those pieces through well-defined interfaces. Consequently, the application development process has been re-engineered such that the software construction is achieved through a component selection, evaluation, and assembly process. The components are acquired from a diverse set of sources, and used together with locally developed software to construct a complete application [Brown96].

Although the concept of designing and implementing software systems as set of components is nothing new, the renaissance in the component-based approach recently are driven by a number of factors and technology advances including:

- The development of the World Wide Web (WWW) and the Internet have increased understanding and awareness of distribute computing. The WWW encourages users to consider systems to be loosely coordinated service that reside "somewhere in hyperspace". In accessing information it becomes unimportant to know where the information physically resides, what underlying engines are being used to query and analyze the data, and so on.
- Both the use of object-oriented software design techniques and languages, and the move from mainframe-based systems toward client/server computing, lead developers to consider applications systems not as monolithic, but rather as separable, interacting components.
- The economic reality that large-scale software development must take greater advantage of existing commercial off-the-shelf (COTS) software, reducing the amount of new code that is required for each application system.
- The materiality and widespread acceptance of several standard component models that make CBD approach a reality.

One of the promises of component technology is a world in which customized business solutions can be assembled from a set of off-the-shelf business objects. The proposition is that independent software vendors produce specialized components for various business problems or requirements. Enterprise simply select the appropriate components that best match their business needs and assemble them into a working solution. Thus far, there is a fairly rich supply of off-the-shelf, third-party, client-side development components. For the moment, the market for server-side components is still very young. As more and more organizations adopt the server component architecture, the market is likely to mature rapidly. Application software companies are already beginning to implement applications using server components. For example, some e-commerce vendors are beginning to supply individual applications, such as a shopping cart and a credit validation server, as customizable components.

The ultimate goal for component-based system is to achieve plug-and-play assembly. In other words, developers should be able to purchase a component, install it, and integrate it into their system with minimal or no effort. A component model makes this goal possible by standardizing the contracts (interface) by which components interact with each other and their runtime environment. These standard interfaces enable a high level of integration and interoperability. They also enable faster application development and allow the use of visual component assembly tools.

2.2.2 Component Models

A component [BDH98, BBC98] is a reusable software building block: a pre-built piece of encapsulated application code that can be combined with other components and with handwritten code to rapidly produce a custom application. A component model [Lyon20] defines the basic architecture of a component, specifying the structure of its interfaces and the mechanism by which it interacts with its container (which provides an application context for one or more components and provides management and control services for the components), and with other components. Thus, the component model provides guidelines to create and implement components that can work together to form a large application. Accordingly, in order to qualify as a component, the application code must provide a standard interface, whose structure of the interface is defined by the component model that enables other parts of the application to invoke its functions and to access and manipulate the data within the component. Application developers can combine components from different developers or different vendors to construct an application.

Currently, there are three most popular and dominant component models [Szyp98a, Szyp98b], namely, Microsoft's (Distributed) Component Object Model (DCOM/COM) [COM] [Micro97, Box98, Micro98w]; Sun's JavaBeans [Sun97j] and Enterprise JavaBeans (EJB) [SunE]; and the Object's Management Group's CORBA [OMG95]. The COM/DCOM is incorporated directly in the Windows operating system. JavaBeans and EJB [SunE] are the component model, and server-side component model for the Java technology [Sun97j], respectively. CORBA is an industry *de facto* standard that provides an object-based middle-ware spanning over heterogeneous platforms and different programming languages. A detail strategic and architectural comparison about these component models may be found in the review papers by Szyperski [Szyp98a], and Plasil and Stal [PS98].

In this thesis, two component models, namely, JavaBeans and CORBA, will be chosen for our infrastructure design. We will introduce CORBA with its naming service, and CORBA transaction services in the next two chapters. In the following, we provide a brief overview of JavaBeans technology, which plays an important role for our Web client tier design and implementation in Chapter 5.

2.2.3 JavaBeans Component Model

JavaBeans is the component model for Java technology. The JavaBeans component model [Sun97j] defines a standard mechanism to develop portable, reusable Java technology components in which there may have features such as events, event registration, properties, and introspection. A JavaBeans component (a bean) actually is a specialized Java class that follows a set of simple naming and design conventions outlined by the JavaBeans Specification [Sun97j].

The JavaBeans component model is commonly regarded as a component model for client-side. Although JavaBeans are widely used for GUI programming that allow developers to visually manipulate components in a IDE builder tool, they can be used equally as an appropriate component model for building server-side programming. In fact, JavaBeans technology will be employed in the implementation of a JSP/Servlet component-based framework for our Web client tier design in Chapter 5.

2.3 Multitier Architecture

The traditional client/server systems have been commonly used to exploit the processing power of workstations to provide more user-friendly and responsive systems. In a traditional client/server application, the client application can contain any or all of the presentation logic (the graphical user interface), application navigation, the business logic (algorithm and business rules), and data manipulation logic (database access). The server is generally a (distributed) database management system, which is actually not a part of the application itself. The traditional client/server architecture is thus also known as a *two-tier architecture*, and this kind of application is called *thick-client* application. Originally, the client/server application promised improved scalability and functionality. Unfortunately, the complexity of delivering enterprise information system services directly to every user and the administrative problems caused by installing and maintaining business logic on every user machine have proved to be major limitations [Lin99]. However, these two-tier limitations may be avoided with the use of multitier architecture.

In a multitier architecture, the application is partitioned and deployed onto three or more interacting tiers, each providing unique functionality to the application. The front-end client usually contains only presentation logic, which may include some very simple control and plausibility check logic. The business logic and data access logic is spread over one or more separate components running on one or more application servers. These business application components, in turn, access enterprise data on backend systems, usually called data servers. This kind of application is also commonly termed as *thin-client* application.

Although multitier architecture has been around for nearly a decade, relatively few organizations have put them to use. Until recently, most organizations did not feel the pressures that required multitier architecture. The impetus of Web-based computing is driving a growing interest in the use of multitier architecture approach, since Web business applications require *thin-client* application architecture to support massive scalability and to support browser-based clients. Multitier architecture provides a number of significant advantages over traditional client/server architecture. Those advantages may be summarized as follows:

- **Increased Scalability and Performance:** Moving the business and data manipulation logic to servers allows an application to take advantage of the power of multithreaded and multiprocessing systems. Server components can pool and share scarce resources, such as processes, threads, database connections, and network sessions. As system demands increase, highly active components can be replicated and distributed across multiple systems. Although modern client/server systems can easily support hundreds of concurrent users, their scalability has limits. Multitier systems can be built with essentially no scalability limits. If the design is efficient, more or bigger servers can be added to the environment to boost performance and to support additional users. Multitier system can thus scale to support hundreds or millions of concurrent users.

- **Increased Reliability:** A multitier environment can also support many levels of redundancy. Through replication and distribution, a multitier architecture eliminates any bottlenecks or single points of failure. The multitier approach supports high reliability and consistent system availability to support critical business operations.
- **Increased Manageability:** A thin-client application is easier to manage than thick-client applications. Very little code is actually deployed on the client systems. Most of the application logic is deployed, managed, and maintained on the servers. Fixes, upgrades, new versions, and extensions can all be administered through a centralized management environment.
- **Increased Flexibility:** The multitier applications architecture supports extremely flexible application systems. The majority of the application logic is implemented in small modular components. The actual business logic in the components is encapsulated behind an abstract, well-defined interface. The code within an individual component can be modified without requiring a change to the interface. Therefore, a component can be changed without impacting on the other components within the application. Multitier applications can thus easily adapt to reflect changing business requirements.
- **Reusability and Integration:** By the nature of its interface, a component is a reusable software building block. Each component performs a specific set of functions that are published and accessible to any other application through the interface. A particular business function can be implemented once and then reused in another application that requires the function. If an organization maintains a comprehensive library of components, application development becomes a matter of

assembling the proper components into a configuration that performs the required application functions.

- **Multi-Client Support:** Any number of clients can access the same server component through its interface. A single multitier application system can support a variety of client devices, including traditional desktop workstations, Web clients, or more esoteric clients, such as information appliances and personal data assistants.

2.4 Web Application

In this section, we introduce and discuss the Web application architecture, Web client models. After the traditional approaches to the Web applications and their disadvantages are addressed, we present an overview of Servlet and JSP technologies, which, together with the use of JavaBeans technology and MVC design pattern, offer several advantages over the traditional approaches.

2.4.1 Web Application Architecture

Web applications are applications that leverage Web clients, such as Web browsers, Web servers, Infrastructure services, and standard Internet protocols. They also typically leverage existing applications and data from external non-Web applications, data and services. It is important to note that Web server and external services are logical tiers capable of running on the same physical machine, or maybe spread across multiple physical machines. Web application is often interchangeable with the term, such as e-business, or e-commerce in the different sector of industry. Applications that can make use of Web clients have several advantages over traditional client/server based applications. One of the major advantages is that it seems to be almost no limit to the possible uses for Web clients in diverse applications. Another major advantage is that it greatly simplifies application deployment and management since to update an application, a developer only needs to change server-side programs instead of thousands of client-installed applications. Modern Web application architecture usually consists of four tiers:

- Client tier: manages Web pages.
- Web tier: manages computer-human interaction.
- Business tier: hosts the business logic packaged as components.
- Data tier: host the back-end databases.

The client tier simply consists of a Web browser. The approaches, along with the technologies employed in this thesis, to the business tier and data tier, will be discussed in next three chapters. In the following, we will focus on the issues related to the development of Web tier.

2.4.2 Web Client Models

Web clients may be classified as application clients, or HTML clients [IAFE99].

An *application client* interacts with a Web server using HTTP or IIOP. The UI (User Interface) to be displayed on the client is generated by part of the Web application that executes on the client. In response to a request from the client, server returns data. This data is interpreted by the client and displayed. The application client model is used primarily in Intranet environments and has the following advantages:

- The programming model is familiar to Web application developers with a traditional client/server background.
- Applications can be made more like standard windowed application.
- The burden of generating the user interface is distributed to each client, reducing the computational load on the Web server.

The major disadvantage of the application client model is that the client code, which is usual a Java applet as it is now well known and accepted as the best choice, is that it tends to be large and require more time to download and initialize. Although Java JAR technology [Tuto98] may increase performance to some extent, the downloading speedy may still pose an issue to the front-end users. With the advent and use of Java plug-in technology [Plugin], the performance may be improved greatly. Nevertheless, the Java plug-in technology may not be supported on some platforms. For example, up to now, the

Java 2 plug-in technology is still not available for IBM AIX, and Windows 2000 platforms. Deployment may thus be restricted since greater client capability and resources, and maintenance as well, are required.

An *HTML client* interacts with a Web server which using HTTP. The UI to be displayed on the client is generated by part of the Web application that executes on the Web server, which responds to every request from the client with a new HTML page. The new page may be static or dynamically generated HTML page that may contain Java applets, JavaScript, etc. The HTML client model is the most common model used in the Internet today and has several advantages:

- Any browser has the potential to run a Web application that is based upon HTTP and HTML.
- The client part of the Web application is small and downloads quickly.
- The server is able to tailor the HTML returned to the client based on upon client or user attributes.

The major disadvantage of the HTML model is the difficulty of creating highly interactive user interfaces. However, the clever use of small applets providing rich UI elements can overcome this shortcoming. On the other hand, highly interactive user interfaces is not usually required in most cases for Internet application. For this reason, the HTML client model will be adopted in this thesis. In the next two sections, after an overview of the traditional approaches, JSPs/Servlet technologies are introduced, as they, together with JavaBeans technology, will be used for our Web tier design and implementation since they can bring great advantages over the traditional approaches.

2.4.3 Traditional Approaches

An early solution to the challenge in creating interactive Web applications was through the use of CGI (Common Gateway Interface) programming. CGI programs run on a Web server, acting as a middle layer between a request from a Web browser or other HTTP (Hypertext Transfer Protocol) client, and databases or applications on the Web server [Holz99]. If the application is set up appropriately, it can retrieve data from a database server and return the data to the Web browser through the HTTP-CGI mechanism. Applications written to work with CGI are often written in a scripting language, such as the Practical Extraction and Reporting Language (PERL) [Holz99] or a UNIX shell script [Rose94].

With traditional CGI, however, it has significant performance and scalability problems since each new CGI request launches a new process on the server. Even if the CGI program itself may be relatively short, the overhead of starting the process may still dominate the execution time. If too many multiple users access the program concurrently, these processes may consume all of the Web server's available resources and the performance thus slows to a grind. To this end, individual Web server vendors have tried to provide solutions to give access to their own server internals. For example, Microsoft's Internet Service API (ISAPI) and Active Server Pages (ASP), and Netscape's Server APIs (NSAPI). These solutions enabled speedy improvement, they were, however, not good enough as they were web-server specific, and thus were incompatible.

2.4.4 Servlet/JSP Technologies

Servelets [SS99] are protocol and platform independent server-side software component, written in Java [Hall20]. They run inside a Java enabled Web server. Servlets are loaded and executed within the Java Virtual Machine (JVM) of the Web server, in much the same way that Java applets are loaded and executed within the JVM of Web client. Since servlets run inside the servers, they do not need a graphical user interface. In this sense, servlets are faceless objects.

Servlets more closely resemble CGI scripts or programs than applets in terms of functionality. As in CGI programs, servlets can response to user events from an HTML request, and then dynamically construct an HTML response that is sent back to the client. Servlets implement a common request/response paradigm for the handling of the message between the client and the server. The Java Servelt API defines a standard interface for the handling of these request and response message between the client and server.

Servlets are powerful tools for implementing complex application. Written in Java, servelts have access to the full set of Java API's, such as JDBC [Tuto98] for accessing enterprise databases.

HTTP is a "stateless" protocol. There is no build-in support for maintaining contextual information. Servlets provide an outstanding technical solution: the *HttpSession* API, which is a high-level interface built on top of cookies or URL-rewriting. Using sessions in servlets involves looking up the session object associated with the current request, creating a new session object when necessary, looking up information associated with a session, storing information in a session, and discarding completed or abandoned sessions. Finally, if you return any URLs to the clients that references your site and URL-rewriting is being used, you need to attach the session information to the URLs. Using *HttpSession* makes it easy for the developer to maintain and access session information within a servlet. It associates an HTTP client with an HTTP session, and it persists over multiple connections by the same use. For example, Java bookstore uses session tracking to keep track of the books being ordered by a user.

As mentioned above, Servlets are similar to CGI in that can produce dynamic Web content. Servlets, however, have the following advantages over traditional CGI programs:

- Portability and platform independence: Servlets are written in Java, making them portable across platforms and across different Web servers, because the Java Servlet API defines a standard interface between a Servlet and a Web server.
- Persistence and performance: A Servelt is loaded once by a Web server, and invoked for each client request. This means that the Servlet can maintain system resources, like a database connection, between requests. Servlets don't incur the overhead of instantiating a new Servlet with each request. CGI processes typically must be loaded with each invocation.
- Java based: Because Servlets are written in Java, they inherit all the benefits of the Java language, including a strong typed system, object-orientation, and modularity, to name a few.

JavaServer Pages (JSPs) are similar to HTML files, but provide the ability to display dynamic content within Web pages [Hall20]. Using JSP technology, web page developers use HTML or XML tags to design and format the result page. They use JSP tags or scriptlets to generate the dynamic content on the page (the content that changes according to the request). The logic that generates the content is encapsulated in tags and JavaBeans components and tied together in scriptlets, all of which are executed on the server side. This makes it very easy to dynamically generate some HTML on a page, while statically composing other content with an HTML editor. JSP Bean tags provide an alternative, component-centric approach to dynamic page and web application design. JSP technology was developed to separate the development of dynamic Web page content from static HTML page design. The result of this separation means that the page design can change without the needed to alter the underlying dynamic content of the page. This is useful in the development life-cycle because the Web page design do not have to know how to create the dynamic content, but simply have to know where to place the dynamic content within the page.

To facilitate embedding of dynamic content, JSPs use a number of tags that enable the page designer to insert the properties of JavaBean object and script elements into a JSP file. JSP is made operable by having their contents (HTML tags, JSP tags and scripts) translated into a Servlet by the Web server. This process is responsible for translating both the dynamic and static elements declared within the JSP file into Java Servlet code that delivers the translated contents through the Web server output stream to the browser. The Servlet produced as a result of the above process, which performs only once when the JSP page is called at the first time, remains in server memory, until the Web server is stopped, the Servlet is manually unloaded, or a change is made to the underlying file, causing recompilation. Consequently, subsequent calls to the page have very fast response times. Some of the advantages of using JSP technology over other methods of dynamic content creation are summarized as follows:

- Separation of dynamic and static content: This allows for the separation of application logic and Web page design, reducing the complexity of Web site development and making the site easier to maintain.

- Platform independence: Because JSP technology is Java-based and platform independent. JSP can run on any nearly any Web application server. JSP can be developed on any platform and viewed by any browser because the output of compiled JSP page is HTML.
- Component reuse: Using JavaBeans, JSP leverage the inherent reusability offered by the JavaBeans technology. This enables developers to share components with other developers or their client community, which can speed up Web site development.
- Scripting and tags: JSP support both embedded JavaScript and tags. JavaScript is typically used to add page-level functionality to the JSP. Tags provide an easy way to embed and modify JavaBean properties and to specify other directives and actions.

Because JSPs are server-side technology, the processing of both the static and dynamic elements of the page occurs in the server. The architecture of JSP/Servlet-enabled Web site is often referred to as *thin-client* because most of the business logic is executed on the server. As such, JSP/Servlet technologies are becoming a key component in a highly scalable architecture for Web applications.

Since JSPs are compiled into Servlets, theoretically developers could write Servlets to support their Web applications. However, JSP technology was designed to simplify the process of creating pages by separating presentation from business logic, due to a built-in facility for calling JavaBeans component. In fact, JavaBeans components, JSPs and Servlets could be combined together following the well-known MVC design pattern [GHJV95] to provide an attractive alternative to other types of dynamic Web programming that offers platform independence, enhanced performance, separation of logic from display, easy of administration, extensibility, and most importantly, ease to use. MVC design pattern with JavaBeans, JSPs and Servlets will be addressed in more detail in Chapter 5.

3 OMA and CORBA

In this chapter, we provide an overview of Object Management Architecture (OMA) with its reference model, and Common Object Request Broker Architecture (CORBA) with its essential components, and CORBA Naming Service. CORBA technology is an industry *de facto* standard that provides an object-based middle-ware spanning over heterogeneous platforms and different programming languages, and is adopted in this thesis for system infrastructure design to address the issues of business tier.

3.1 Object Management Architecture

As enterprise applications usually require information among a diversity of sources, such as, in-house, brought-in, supplier, customer, etc. Those applications become increasingly difficult and complex, and thus should be built using a methodology that supports modular production of software; encourages reuse of code; allows useful integration across lines of developers, operating systems and hardware; and enhances long-range maintenance of that code. The Object Management Group (OMG) proposed Object Management Architecture (OMA) based on which to develop integrated software system [OMG97]. It has the ability to grow in functionality through the extension of existing components and addition of new components to the system. This results in faster application development, easier maintenance, reduced program complexity, and reusable components. It provides transparency over heterogeneous networks.

The OMA provides the conceptual infrastructure upon which supporting specifications are based. OMG is populating the OMA with detailed specifications for each component and interface category in the Reference Model. The OMA Reference Model [OMG97] is an architectural framework for the standardization of interfaces to infrastructure and services that applications can use [Siegel96]. Adopted specifications include the CORBA, CORBAservices, and CORBA facilities.

3.2 OMA Reference Model

The Reference Model, as depicted in Figure 3.1, identifies and characterizes the components, interfaces, and protocols that compose the OMA. This includes the Object Request Broker (ORB) component that enables clients and objects to communicate in a distributed environment, and four categories of object interfaces, which are briefly described as follows:

- ***Object Request Broker (ORB)*** is situated at the conceptual center of the Reference Model. It acts as a message bus between objects, which may be located on any machine in a network, implemented in any programming language, and executed on any hardware or operating system platform.
- ***Object Services*** component standardizes the life cycle management of objects. Functions are provided to create objects, to control access to objects, to keep track of relocated objects and to consistently maintain the relationship between groups of objects. The published services include Naming, Transaction, Events, Life Cycle, Externalization and Licensing, etc.
- ***Common Facilities*** are interfaces for horizontal end-user-oriented facilities applicable to most application domains. It provides a set of generic application functions that can be configured to the requirements of a specific configuration. Examples are printing facilities, database facilities, and electronic mail facilities.
- ***Domain Interfaces*** are application domain-specific interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecom, Electronic commerce, and Transportation.
- ***Application Interfaces*** are interfaces developed specifically for a given application, which is part of the architecture, represents those application objects performing specific tasks for users.

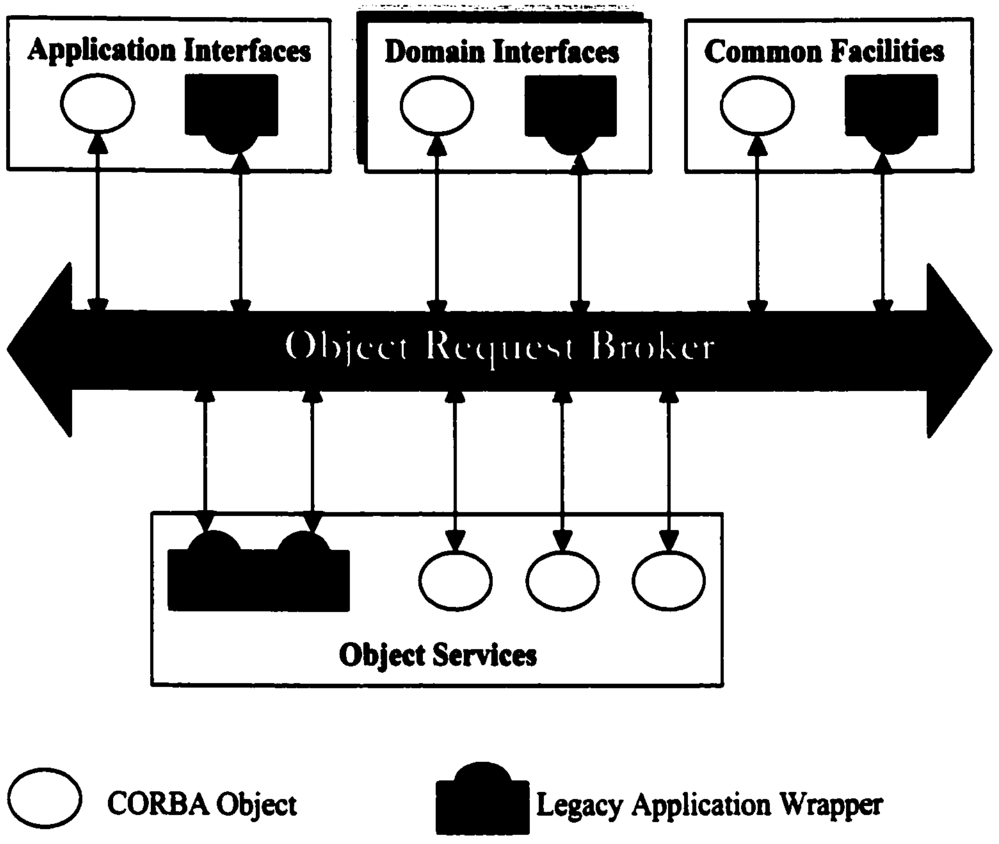


Figure 3.1 OMA Reference Model

We note that the Objects and Legacy Application Wrappers are represented as different shapes in the figure, to highlight the capability of CORBA in incorporating legacy code.

The applications need only support or use OMG-compliant interfaces to participate in the OMA. In OMA, interfaces are specified in OMA Interface Definition Language (IDL) [Siegel96]. The OMA IDL provides a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Almost all-major programming languages have their mapping to CORBA IDL. They include C, C++, Smalltalk, COBOL and Java [Siege96].

3.3 Overview of CORBA

The Object Request Broker component of the Object Management Architecture is the communication heart of the standard. This is referred to commercially as CORBA (Common Object Request Broker Architecture) [OMG95]. CORBA specifications include a language to define objects, a protocol for exchanging information and a protocol for passing objects over the distributed systems. It provides a platform, location and an implementation-language-neutral architecture for the development of distributed applications. As a result, currently CORBA has become one of the most commonly used architectures for the development of distributed Web-enabled applications.

CORBA uses an object-oriented approach for creating software components that can be reused and shared among applications. Each object encapsulates the details of its inner workings and presents a well-defined interface, which reduces application complexity. The cost of developing applications is also reduced, because once an object is implemented and tested, it can be used over and over again.

CORBA fits the component-based and Internet-based approaches to building and using software. It defines a way to divide application logic over objects distributed over the network. CORBA also defines a way for these objects to communicate with one another and to use each other's services. It also manages how objects identify themselves, find others, learn about network events, handle object-to-object transactions, and maintain the object security.

3.3.1 Essential Components of CORBA

CORBA is an Object Request Broker framework designed to manage objects from heterogeneous worlds. Figure 3.2 illustrates the primary components in the CORBA architecture. A brief summary of these components is as follows:

- ***Object Implementation*** – This defines operations that implement a CORBA IDL interface. Object Implementation can be written in a variety of languages including C, C++, Java, Smalltalk, and Ada [Siegel96].

- ***Client*** – This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller.
- ***Object Request Broker (ORB)*** – The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations.
- ***ORB Interface*** – This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface.
- ***CORBA IDL stubs and skeletons*** – CORBA IDL stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB. The stub and skeleton code combine with the ORB to implement location transparency.
- ***Dynamic Invocation Interface (DII)*** – This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. The DII also allows clients to make non-blocking deferred synchronous and oneway calls.
- ***Dynamic Skeleton Interface (DSI)*** – This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation, that does not have compile-time knowledge of the type of the object it is implementing.
- ***Object Adapter*** – This assists the ORB with delivering requests to the object and with activating the object. The object adapter associates object implementations with the ORB. Currently CORBA defines the Basic Object Adapter (BOA) and the Portable Object Adapter (POA). Their purpose is to generate and interpret object references, and to activate and deactivate object implementations.
- ***GIOP/IIOP*** – The General Inter-ORB Protocol (GIOP) defines a linear format for the transmission of CORBA requests and replies without requiring a particular network transport protocol. The Internet Inter-ORB Protocol (IIOP), which applies to TCP/IP networks, is a specialization of GIOP. The IIOP lets a client make a request to one

ORB and have the request relayed through a different ORB to the server object. IIOP leaves you free to choose different ORB's for different parts of your application, and it lets anything that implement IIOP act as a CORBA server or client.

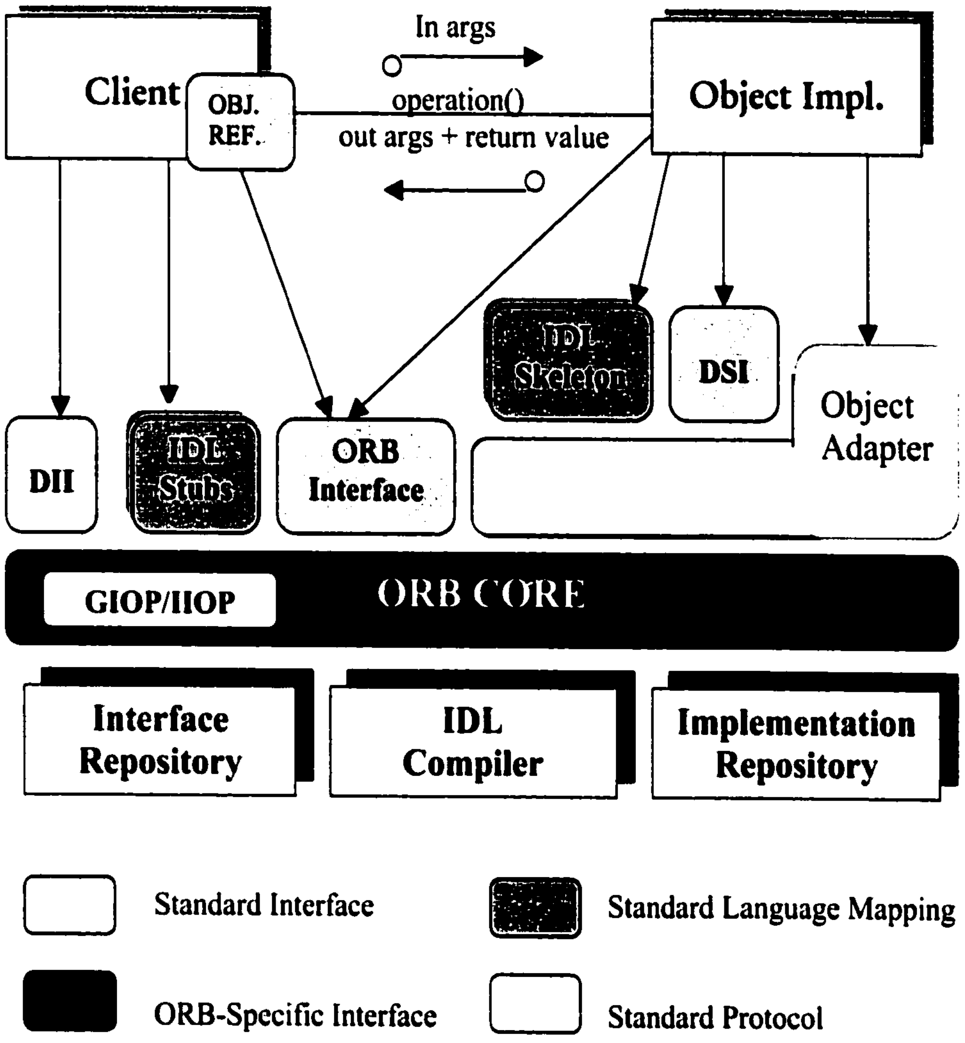


Figure 3.2 CORBA Architecture with its Essential Elements

3.3.2 CORBA Naming Service

In addition to the interoperability offered by the ORB, CORBA also defines a set of system-level services that complement and augment the functionality of the ORB. These are called CORBAservices, and there are currently standard IDL interfaces defined for over fifteen CORBAservices. These include services for life cycle management, persistence, naming, transactions, queries, events, named properties, concurrency control, licensing and security. In the following, we present a brief overview of CORBA Naming Service. Another most important CORBAservice, CORBA Transaction Service, which is adopted in our system infrastructure, along with comparison with other available transaction management technologies, however, will be detailed in the next chapter.

CORBA Naming Service is one of OMG's common object services. The Naming Service provides the capability for CORBA objects to find other CORBA objects using an easily distinguished naming convention. The Naming Service also enables us to associate a logical name with an object at runtime, which provides more flexibility in configuring your distributed object system.

The Naming Service provides the ability to bind a name to an object relative to a naming context. A naming context is an object that contains a set of name bindings in which each name is unique. To resolve a name is to determine the object associated with the name in a given context. Through the use of a very general model and in dealing with names in their structural form, Naming Service implementations can be application specific or be based on a variety of naming systems currently available on system platforms. The principal task of the Naming Service is to keep track of the namespace, which is the collection of object names bound to a Naming Service. The namespace may contain a hierarchy of bindings. The binding is the logical association of the object reference to its symbolic name.

A client obtains a reference to the Naming service by calling the *resolve_initial_references()* method of the ORB. Once the initial context of the Naming Service is resolved, it calls the *resolve()* method on the NamingContext to obtain a reference to the desired object. The whole task of name resolution is transparent to the client.

CORBA Naming Service is defined in the CosNaming Module which consists of two interfaces are as follows:

- The ***NamingContext*** interface contains operations for managing and manipulating object names and naming contexts, such as, *bind(), rebind(), unbind(), bind_context(), new_context(), bind_new_context(), resolve()*, etc.
- The ***BindingIterator*** interface enables clients to navigate through a set of bindings found in a naming context.

In summary, CORBA Naming Service can be used to assign names logically to objects in a naming context and to find these objects easily, based on their compound names. The location services can be configured at runtime and can be helpful in locating objects in complex distributed systems.

4 Distributed Transaction Management

Transaction management is one of the most crucial requirements for enterprise application development [Subr99] [Jim93]. Most of the large enterprise applications in the domains of finance, banking and electronic commerce rely on transaction processing for delivering their business functionality [GN96]. Given the complexity of today's business requirements, transaction processing occupies one of the most complex segments of enterprise level distributed applications to build, deploy and maintain. In this chapter, we present an introduction to the concepts involved in distributed transaction management. Several available Transaction Services, in particular, VisiBroker's Integrated Transaction Service (ITS), one of concrete implementation of CORBA Transaction Service, adopted in this thesis for our system infrastructure design, are also discussed.

4.1 Concept of Transaction

Enterprise applications often require concurrent access to distributed data shared amongst multiple components, to perform operations on data. A ***transaction*** is also a unit of consistency, *i.e.,* executed from a coherent database state; the transaction drives the database to another coherent state. A transaction ends either by a *commit* (the transaction's effects become permanent in the database), or by an *abort* (the effects are cancelled) [Cou96].

Applications should maintain integrity of data under the following circumstances: distributed access to a single resource of data, and access to distributed resources from a single application component. In such cases, it may be required that a group of operations on (distributed) resources be treated as one unit of work. In a unit of work, all the participating operations should either succeed or fail and recover together. This problem is more complicated when

- A unit of work is implemented across a group of distributed components operating on data from multiple resources, and/or

- The participating operations are executed sequentially or in parallel threads requiring coordination and/or synchronization.

In either case, it is required that success or failure of a unit of work be maintained by the application. In case of a failure, all the resources should bring back the state of the data to the previous state (*i.e.*, the state prior to the commencement of the unit of work).

The concept of a transaction, and a transaction manager (or a transaction processing service) simplifies construction of such enterprise level distributed applications while maintaining integrity of data in a unit of work. By definition, a ***transaction*** is *a unit of work* done by multiple distributed components on shared data, ensuring the following properties [Little98]:

- ***Atomicity***: A transaction should be done or undone completely and unambiguously. In the event of 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 (such as integrity constraints) defined on the data. On completion of a successful transaction, the data should be in a consistent state. In other words, a transaciion should transform the system from one consistent state to another consistent state. For example, in the case of relational databases, a consistent transaction should preserve all the integrity constraints defined on the data.
- ***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 serially should be the same as that of running them concurrently.
  - During the course of a transaction, intermediate (possibly inconsistent) state of the data should not be exposed to all other transactions.
  - Two concurrent transactions should not be able to operate on the same data.
- ***Durability***: When a transaction commits, its effects cannot be cancelled by the execution of an uncommitted transaction.

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

4.2 Distributed Transactions

A *Distributed Transaction* is a client transaction that invokes operations in several different servers [Cou96]. A client transaction may involve one or multiple servers. Transaction activities spread into multiple servers either directly by requests made by a client or indirectly via requests made by servers.

4.2.1 Classification of Distributed Transactions

Distributed transactions can be structured in two different ways: as simple distributed transactions and as nested distributed transactions [Cou96].

- Simple distributed transactions - A client makes requests to more than one server, but each server carries out the client's requests without invoking operations in other servers.

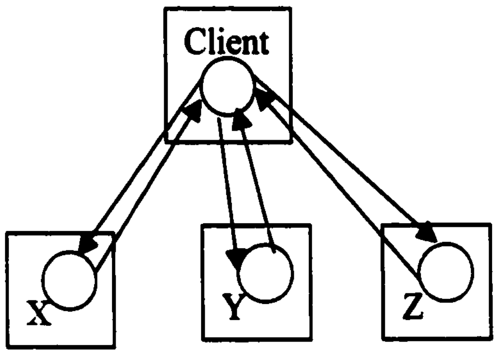


Figure 4.1 Simple Distributed Transaction

Figure 4.1 shows transaction T is a simple distributed transaction that invokes operations in servers X, Y and Z. A simple (not-nested) client transaction completes each of its requests before going on to the next one. Therefore each transaction accesses servers' data items sequentially.

- Nested distributed transactions - A client makes requests to more than one server, in some cases an operation in a server may invoke an operation in another server and in general the latter may invoke further operations in yet more servers and so forth. In this situation, each client transaction is structured as a set on nested transaction.

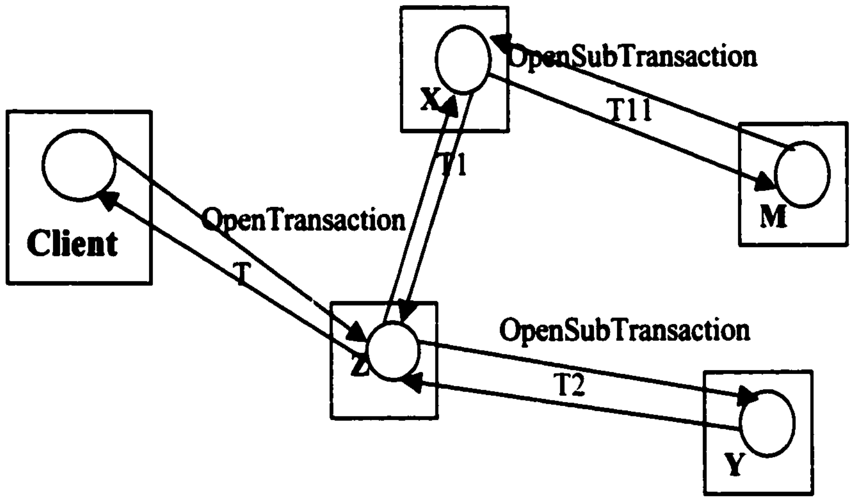


Figure 4.2 Nested Distributed Transaction

Figure 4.2 shows a client's transaction T at server Z invoking operations in servers X and Y, which form nested transactions T1 and T2. The nested transaction T1 invokes operations in server M, which form further nested transaction T11. In general, a transaction consists of a hierarchy of nested transactions. Nested transactions at the same level may run concurrently with one another.

4.2.2 Distributed Transaction Protocols

When a distributed transaction comes to an end, the atomicity property of transactions requires that either all of the servers involved *commit* the transaction or all of them *abort* the transaction. To archive this, one of the servers takes on a coordinator role, which is responsible for aborting or committing transaction and adding other servers. The manner in which the coordinator archives this depends on the protocol chosen. There are two atomic commit protocols:

- ***One-phase atomic commit protocol*** – The coordinator communicate the commit or abort request to all of the servers in the transaction and to keep on repeating the request until all of them had acknowledged that they had carried it out.
  As in the case when the client requests a commit, simple one-phase atomic commitment protocol does not allow a server to make a unilateral decision to abort a transaction. The client may not know when a server has failed and restarted during the progress of a distributed transaction - such a server will need to abort the transaction. So the simple one-phase atomic commitment protocol is inadequate.

- ***Two-phase atomic commit protocol*** – In the first phase, each server has voted to commit a transaction, it is not allowed to abort it. In the second phase, if any one server votes to abort, the decision must be to abort the transaction; if all the servers vote to commit, the decision must be to commit the transaction.
  The two-phase commit protocol is designed to allow any server to abort its part of a transaction. Due to atomicity, if one part of a transaction is aborted, then the whole transaction must also aborted.

4.3 Distributed Transaction Management Standards

In this section, we present an overview of several available transaction services with focus on VisiBroker's Integrated Transaction Service (ITS), involved in the distributed transaction management.

4.3.1 Microsoft Transaction Server

Microsoft Transaction Server (MTS) is a component-based transaction server for components based on the Microsoft's Component Object Model (COM). The MTS programming model provides interfaces for building transactional COM components, while the MTS runtime environment provides a means to deploy and manage these components and manage transactions. Using the MTS, work done by multiple COM components can be composed into a single transaction.

But MTS is only supported on Windows 95 and NT platform, it is not platform independent and is not based on open specifications.

4.3.2 X/Open Distributed Transaction Processing (DTP) Model

In the distributed transaction management domain, X/Open Distributed Transaction Processing (DTP) model, proposed by the Open Group, is the most widely adopted model for building transactional applications. This model is a standard among most of the commercial vendors in transaction processing, databases domain, and message queuing.

This model defines four components: *application programs, resource managers*, a *transaction manager*, *communication resource manager* (facilitate interoperability between different transaction managers in different transaction processing domains). This model also specifies functional interfaces between application programs and the transaction manager (known as the *TX interface*), and between the transaction manager and the resource managers (the *XA interface*). With products complying with these interfaces, one can implement transactions with the two-phase commit and recovery protocol to preserve atomicity of transactions.

4.3.3 OMG Object Transaction Service

Object Transaction Service (OTS) is another distributed transaction processing model specified by the Object Management Group (OMG) [OMG95]. This specification extends the CORBA model and defines a set of interfaces to perform transaction processing across multiple distributed CORBA objects. The OTS model is based on the X/Open DTP model [Little99] with the following enhancements:

- The OTS model replaces the functional XA and TX interfaces with CORBA ***IDL*** interfaces.
- The various objects in this model communicate via CORBA method calls over ***IIOP***.

However, the OTS is interoperable with X/Open DTP model. An application using transactional objects could use the TX interface with the transaction manager for transaction demarcation.

The OTS architecture consists of the following components: *Transaction Client, Transactional Object, Recoverable Object, Transactional Server, Recoverable Server,* and *Resource Object*, which described in Figure 4.3. The Transaction Service defines interfaces that allow multiple, distributed objects to cooperate to provide atomicity. These interfaces enable the objects to either commit all changes together or to rollback all changes together, even in the presence of (noncatastrophic) failure. No requirements are placed on the objects other than those defined by the Transaction Service interfaces. Objects supporting transactional behavior must have interfaces derived from the ***TransactionalObject*** interface.

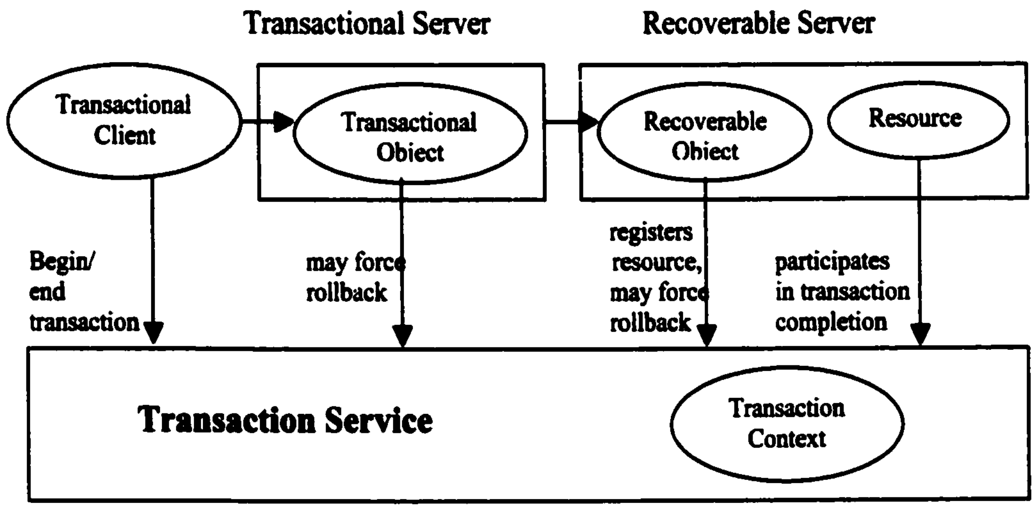


Figure 4.3: Object Transaction Service Entities

In addition to the usual transactional semantics, the CORBA OTS provides application objects to synchronize transient state and data stored in persistent storage.

The OTS provides interfaces that allow multiple distributed objects to cooperate in a transaction such that all objects commit or abort their changes together. As an improvement to the X/Open reference model, the OTS is fully compatible with X/Open compliant software. Thus the distributed transaction processing reference standard has been upgraded to an object-oriented model, promoting software component reuse, and interprocess communication mechanisms have been cleanly defined, facilitating a common standard for vendor interoperability.

4.3.4 Java Transaction Service

Java^{TM} Transaction Service (JTS) [JTS99] specifies the implementation of a Transaction Manager which supports the Java^{TM} Transaction API (JTA) 1.0 Specification at the high-level and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 Specification [OTS97] at the low-level. JTS uses the standard CORBA ORB/TS interfaces and Internet Inter-ORB Protocol (IIOP) for transaction context propagation between JTS Transaction Managers.

A JTS Transaction Manager provides transaction services to the parties involved in distributed transactions: the application server, the resource manager, the standalone transactional application, and the Communication Resource Manager (CRM). The packages ***org.omg.CosTransactions*** and ***org.omg.CosTSPortability*** define the Java mapping of the OMG's Transaction Service (OTS) 1.1 interfaces using the standard IDL/Java programming language mapping.

Although the Java Transaction Service (JTS) [JTS99] is a Java implementation of the OMG OTS 1.1 specification, the Java Transaction API (JTA) 1.0 retains the simplicity of the XA and TX functional interfaces of the X/Open DTP model [Little98].

In the Java transaction model, the Java application components can conduct transactional operations on JTA compliant resources via the JTS. The JTS acts as a layer over the OTS. The applications can therefore initiate global transactions to include other OTS transaction managers, or participate in global transactions initiated by other OTS compliant transaction managers.

4.3.5 VisiBroker Integrated Transaction Service (ITS)

VisiBroker's Integrated Transaction Service (ITS) is INPRISE's implementation of the CORBA Transaction Service [VITS99]. Implemented on top of Inprise's VisiBroker ORB, ITS combines the functionality of several services – including a transaction service that is compliant with CORBA's Transaction Service version 1.1 specification, database and legacy system – into one, integrated architecture. It provides reliable, high-performance transactional integrity and enables distributed applications to handle the various difficulties of transaction completion over the Internet and Intranets [VITS99]. Implementation language independence provided by ITS offers more flexibility than JTS, which can only be implemented by Java language.

4.3.5.1 VisiBroker Integrated Transaction Service Specification: IDL Interfaces

VisiBroker ITS defines IDL interfaces that allow multiple distributed objects to participate in a transaction, and enable a distributed application to handle transaction completion over the Internet and Intranets. The interfaces of the two primary components in Integrated Transaction Service have been defined with IDL in the modules of ***CosTransactions*** and ***VISTransactions***. The following IDL code segments are part of these specifications. This will be served as a starting point for the implementation discussion in the following sections. Interfaces in the ***VISTransactions*** module inherit from and extend the ***CosTransactions*** interfaces.

```
module CosTransactions
{
    exception Unavailable {};
    interface Control
    {
        Terminator get_terminator() raises(Unavailable);
        Coordinator get_coordinator()raises(Unavailable);
    };
    ......
}

module VISTransactions
{
    interface Current : CosTransactions::Current
    {
        void begin_with_name (in string userTransactionName)
            raises(CosTransactions::SubtransactionsUnavailable);
        ......
    };
    ......
}
```

4.3.5.2 Models for Transaction

The ITS Transaction Service works with objects at the ORB level to coordinate and manage a transaction's commit or rollback. The ORB enables the ITS Transaction Service to propagate the *transaction context* to each object participating in the transaction. This section we will describe the Models for ITS transaction completion. The Model for transaction completion with Resource objects expands on the model for basic transactions when transactions with Resource and Recoverable Server.

4.3.5.2.1 Model for Basic Transactions

The ITS Transaction Service interacts with an application when the transaction originator begins the transaction, as transactional information is propagated to transactional objects, and finally, coordinates the transaction's completion (commit or rollback) across multiple objects.

- *Beginning the transaction*

  First when the transaction originator registers with the ITS Transaction Service its desire to begin a transaction. The ITS Transaction Service answers this request by returning a transaction context to the transaction originator. The transaction context is then associated with the originator's thread of control that was issued by the VisiBroker ORB.

- *Issuing requests to transactional objects*

  As the transaction originator issues requests to transaction objects, each of these requests is also associated with the transaction context. The ITS Transaction Service propagates the transaction context to all objects participating in the transaction via ORB.

- *Completing a transaction*

  If a commit is requested and all participating Resources agree to commit, then the changes are committed. If any participant votes for rollback, then the transaction is rollback. The ITS Transaction Service will rollback the transaction when the timeout period expires.

4.3.5.2.2 Model for Transaction with Resource Objects

This kind of model expands on the basic model to show the objects that necessary when transactions involve data. If a transactional application only involves one Resource, the ITS Transaction Service initiates a one-phase commit process, and if a transactional application involves more than one Resource, the ITS Transaction Service initiates a two-phase commit process. We explain the process of two-phase commit in the following section.

- *Resource object is registered for the transaction*

  Resource objects must be registered for all recoverable data involved in the transaction. The transactional object registers the Resource with the transaction's Coordinator for the recoverable data.
- *Transaction originator initiates transaction completion*

  The transaction originator notifies the Terminator that it wishes to complete the transaction, which initiates the two-phase commit process with the ITS Transaction Service.
- *Terminator tells Resource object to prepare*

  Once the Terminator receives notice that the transaction originator wishes to commit the transaction, the Terminator contacts all Resource objects participating in the transaction, and notifies them they must prepare to commit the transaction.
- *Resource objects return a vote to the terminator*

  When Resource objects are told to prepare, they respond to the Terminator with a vote (VoteCommit or VoteRollback or VoteReadOnly). If the Resource returns VoteRollback or VoteReadOnly, it will not be contacted again by the ITS Transaction Service, and can safely destroy itself.
- *Terminator decides whether to commit or rollback*

  Based on the votes received by the Resource objects, the Terminator determines whether the transaction will be committed or rolled back. If the transaction decision was to rollback, the Terminator invokes *rollback()* on all Resources. If the decision is to commit, the Terminator invokes *commit()* on all Resources, and the two-phase commit process is finished.
- *Resource objects vote to commit the transaction*

  When a Resource object commits a transaction, it makes any data changed by the transaction visible to all readers of the data – the data stored by the recoverable object is changed according to the outcome of the transaction. Also, the Resource object stores other information in case of failure. Lastly, once the transaction has been committed, all objects associated with the transaction are removed (i.e. the Coordinator, Terminator, and Recovery Coordinator).

To conclude this section, we provide a brief summary of the steps for one-phase commit, two-phase commit, and rollback as follows:

- **One-phase commit**
  - Resource objects are registered for the transaction
  - Transaction originator initiates transaction completion
  - Terminator tells Resource objects to prepare
  - Resource objects return a vote to the Terminator
  - The Terminator decides whether to commit or rollback
  - The Terminator tells Resource objects to commit or rollback
- **Two-phase commit**
  - Resource objects are registered for the transaction
  - Transaction originator initiates transaction completion
  - Terminator tells Resource objects to commit one phase
  - Resource objects return a vote to the Terminator
  - The Terminator decides whether to commit or rollback
  - The Terminator tells Resource objects to commit or rollback
- **Rollback**
  - Resource objects are registered for the transaction
  - Transaction originator initiates transaction completion
  - Terminator tells Resource objects to rollback

4.3.5.3 The Advantages of VisiBroker's ITS

VisiBroker ITS manages the completion of transactions using a one-phase or two-phase commit protocol. In addition to the transaction management features of CORBA's Transaction Service, VisiBroker ITS has numerous import features:

- Transaction service built from the ground up with modern technology, tightly integrated with standard backbone for the emerging Web infrastructure
- Built specifically for transactions in a distributed environment
- Truly distributed architecture that does not require a transaction service on each node

- Flexibility to deploy transaction service on the appropriate number of servers to meet system requirements
- Fully scalable with load balancing, connection management, and multithreading from the underlying ORB
- Complete Java solution-supports both Java server objects and Java clients
- Compliance with CORBA and other OMG standards
- Interoperable with other CORBA-compliant software, ORBs, and CORBA Transaction Services
- Easy to install and configure

The power and flexibility offered by the VisiBroker ITS solution is based on its distributed architecture. Unlike other transactional solutions, VisiBroker ITS was designed to support dynamic objects in a distributed environment. The ITS Transaction Service and the corresponding transaction logs are not required on every server but can be dynamically discovered by the VisiBroker ORB.

VisiBroker ORB, VisiBroker ITS simplifies the complexity of distributed transactions by providing an essential set of services-including an implementation of the CORBA Transaction Service, recovery and logging services, integration with databases and legacy systems, and administration facilities-within a single, integrated architecture.

5 System design and Implementation

In this chapter, we provide an in-depth description of a prototype development for a distributed, multitier, Web-based transaction system architecture design and implementation. We review and illustrate the entire process through developing a sample Web-based transaction system – an online bookstore. The sample application system architecture design is based on the principles discussed in the earlier chapters, and implemented by the use of two component models, CORBA and JavaBeans, together with the platform-independent technologies, such as, JSPs/Servlets, JDBC, and Java programming language. The MVC design pattern and UML modeling are also employed in our system design.

5.1 Requirements Analysis

Our Web transaction system is designed for multitier applications, and offers a lot of flexibility in choosing how to distribute application functionality across the tiers. As a web-enabled application, the client tier provided by the browser, the Web tier and database tier, which holds persistent application data, are provided by the server. The requirement of our Component-based Web Transaction Systems (CWTS) running environment is listed as following:

- **Client Side**

  Any machine which installed Web browser

- **Server Side**

  Hardware:

  - A PC with Pentium II CPU or equivalent, 96 megabytes RAM minimum, the operating system is Windows NT 4.0 Service Pack 6.
  - A 28.8kbps modem or LAN network card connected to Internet.

  Software:

  - Java 1.1.x platform runtime.
  - VisiBroker ITS 1.1 (including VisiBroker 3.3 for Java).

- JavaServer™ Web Development Kit 1.0.1.
- Oracle8I, Database Manage System.

5.2 System Design Goals and Considerations

The issue to be addressed in the development of the overall system architecture is object decomposition. While these requirements apply to object-oriented design in general, they become even more important for multitier enterprise applications. The framework must enables:

- Reuse of software designs and code
- Identification of responsibility of each object. The division into objects must ensure that the responsibilities of each object – what the object represents and what must it accomplish – are easily and unambiguously identified.
- Separate of stable code from more volatile code – All parts of an enterprise application are not equally stable. The parts that deal with presentation and user interface change more often. The business rules and database schemas employed in the application have a much lower propensity to change.
- Divide development effect along skill lines – The decomposition should result in a set of objects that can be assigned to various subteams based on their particular skills.

As a growing number of sophisticated services are provided over the Internet, the need arise to coordinate the activity of multiple objects into transactions to ensure consistency, isolation, and durability. Facing with the practical limits of two-tier computing and static three-tier environments, distributed-object architectures require a transaction-process solution that not only delivers the features of tradition TP (Transaction Processing) Monitors, but also meets the challenges of today's heterogeneous computing environments. So our additional objectives are:

- Support for industry-endorsed standards, such as CORBA, Java, and IIOP
- High performance and scalability in distributed computing environments
- Support for object-oriented application

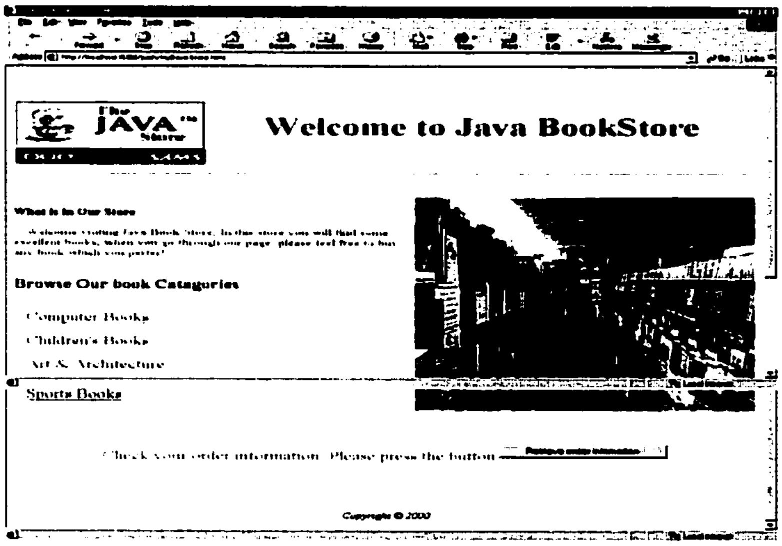
- Reliable, fast, transaction-safe access to multiple data sources
- Support for multiple platforms and programming languages
- Security and administration for distributed environments

5.3 System Architecture

The discussion turns to the architecture of the prototype: the partitioning of functionality into modules, the assignment of functionality to tiers, and object decomposition within tiers. And also goes to build and deploy robust distributed object to participate in coordinated transactions, and enables a distributed application to handle the various impediments to transaction completion over the Internet and intranets.

5.3.1 A Typical Web Site

In a Web browser, the user enters a URL, that points to a page on the package delivery company's Web server. The Web server receives the user request and uses HTTP to return a Web page to the client's browser. The following figure shows a typical web site of a sample on-line e-commerce.



Welcome to Java BookStore

What is in Our Store

[illegible]

Browse Our book Categories

Computer Books

Children's Books

Art & Architecture

Sports Books

Check your order information. Please press the button

Copyright © 2000

Figure 5.1: the homepage of Java Bookstore

5.3.2 System Model – Distributed, Multitered, Component-Based Model

As explained in the previous chapters, Multitier Distributed Component-based Model provides more advantage over traditional client-server model (Thick Client).

Our approach system adopts a multitier distributed application model. The whole application system is divided into components based on functionality. Various parts of an application can run on different devices. Different components are installed to make up an integrated system. The system architecture defines *a client tier, a web tier, business tier and database server tier.* The client tier supports a variety of client types, both outside and inside of corporate firewalls. The Web tier is made up of JavaBeans, Servlets/JSPs generating dynamic HTML pages. The business tier handles business logic, such as accessing database with CORBA and JDBC.

- A Web browser runs as a front-end to the application in the ***Client Tier*** and downloads static/dynamic HTML pages generated by Servlets/JSPs from web server.
- ***Web Tier*** is made up of Servlets/JSPs that generate dynamic HTML pages, and JavaBeans used in JSP for dispatching the user input to ***Business tier*** for processing separation of presentation and business logic.
- CORBA objects process distributed transaction logic via VisiBroker ITS in ***Business Tier***.
- CORBA object access ***Database Server Tier***, which Oracle8I Database Server reside in, through Oracle JDBC driver.

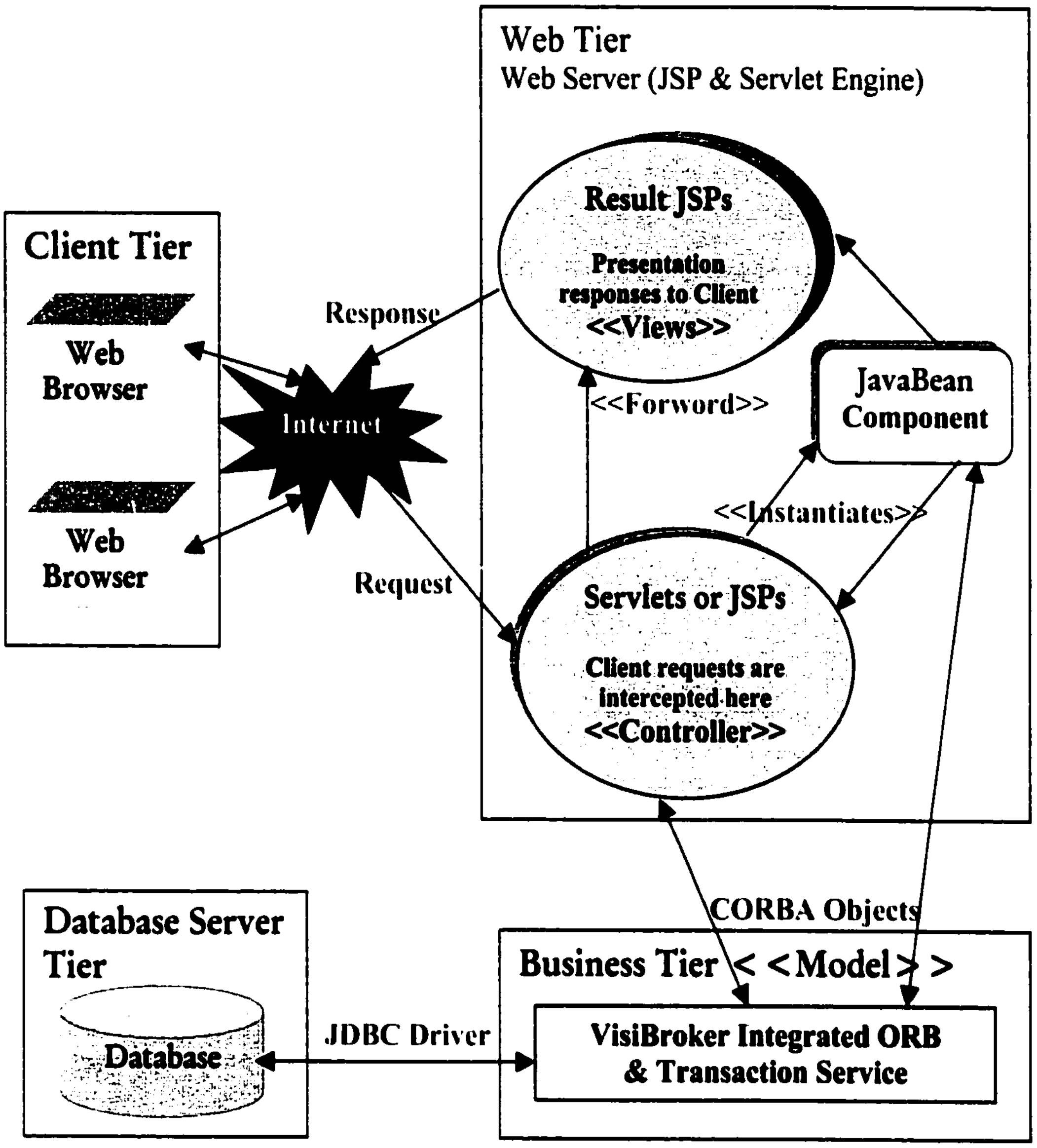


Figure5.2: System Architecture

Figure5.2 shows: front-end components (Servlets and JSPs) accept a request, and then forward the information to other JSP file. The presentation component (JSP page) processes the request and returns the response directly to the user. The front-end components (Servlet file and JSPs) act as the controller and are in charge of the request processing and the creation of any JavaBeans or objects used by the presentation component (JSP page) the view. There is no processing logic within the presentation JSPs themselves; they are simply responsible for retrieving any objects or beans that may have been previously created by the JSPs, and extracting the dynamic content for insertion within static templates.

The Web tier is responsible for almost all of the application functionality. It must take care of dynamic content generation and presentation and handling of user requests. It must implement core application functionality such as order processing and enforce business rules defined by the application. The web tier only used as a front end for receiving client Web requests and for presenting HTML responses to the client.

In the Business tier, the ITS Transaction Service interacts with an application when the transaction originator begins the transaction, as transactional information is propagated to transactional objects, and finally, coordinates the transaction's completion (commit or rollback) across multiple objects.

The JDBC DirectConnect driver from VisiBroker ITS enables ITS transactional applications to use Oracle JDBC driver for database connectivity with Oracle database in database Server tier.

5.3.3 The Architecture of the Prototype

Partitioning the application into logical modules is the first step in subdividing the overall problem, The next step is to begin the process of object-oriented design of the application, identifying units of business logic, data, and presentation logic and modeling each of them as a software object.

This section describes the architecture of the Java BookStore application; exploring the partitioning of functionality into modules, the assignment of functionality to tiers, and object decomposition within the tiers. Dividing the application into modules based on similar or related functionality reduces the dependency between modules, allowing them to be developed independently. The sample application demonstrates an approach that started out simple and small, but kept the option of growth open.

The functional modules could be identified as the following modules and their corresponding responsibilities:

- ***Customer Account*** module – The application tracks user account information saves user account information to a database so that it spans sessions.
- ***Product Catalog*** module – The application allows the user to search for products or services and be able to display details of individual books.
- ***Order processing*** module – The application performs order processing. Order processing occurs when the user performs the checkout process and buys the items in the shopping cart.
- ***Customer Order*** module – The application performs the customer order process and save into the database in order for later retrieve.
- ***Inventory*** module – The application maintains information on the number of each type of book in stock
- ***Messaging*** module – The application sends confirmation messages.
- ***Control*** module – The application allows users to browse the book catalog and add selected items to a shopping cart. At any time, the user can modify items in the shopping cart, add new items, or remove items already placed in the cart.
- ***Transaction Processing*** module – The application coordinate several database table updated in a single atomic operation and ensure the data integrity in distributed transaction environment.
- ***Data Processing*** module – The application retrieve, update item quantity in Oracle database with ITS JDBC DirectConnect driver and Oracle JDBC driver.

Figure 5.3 shows the interrelationship of the modules in the application.

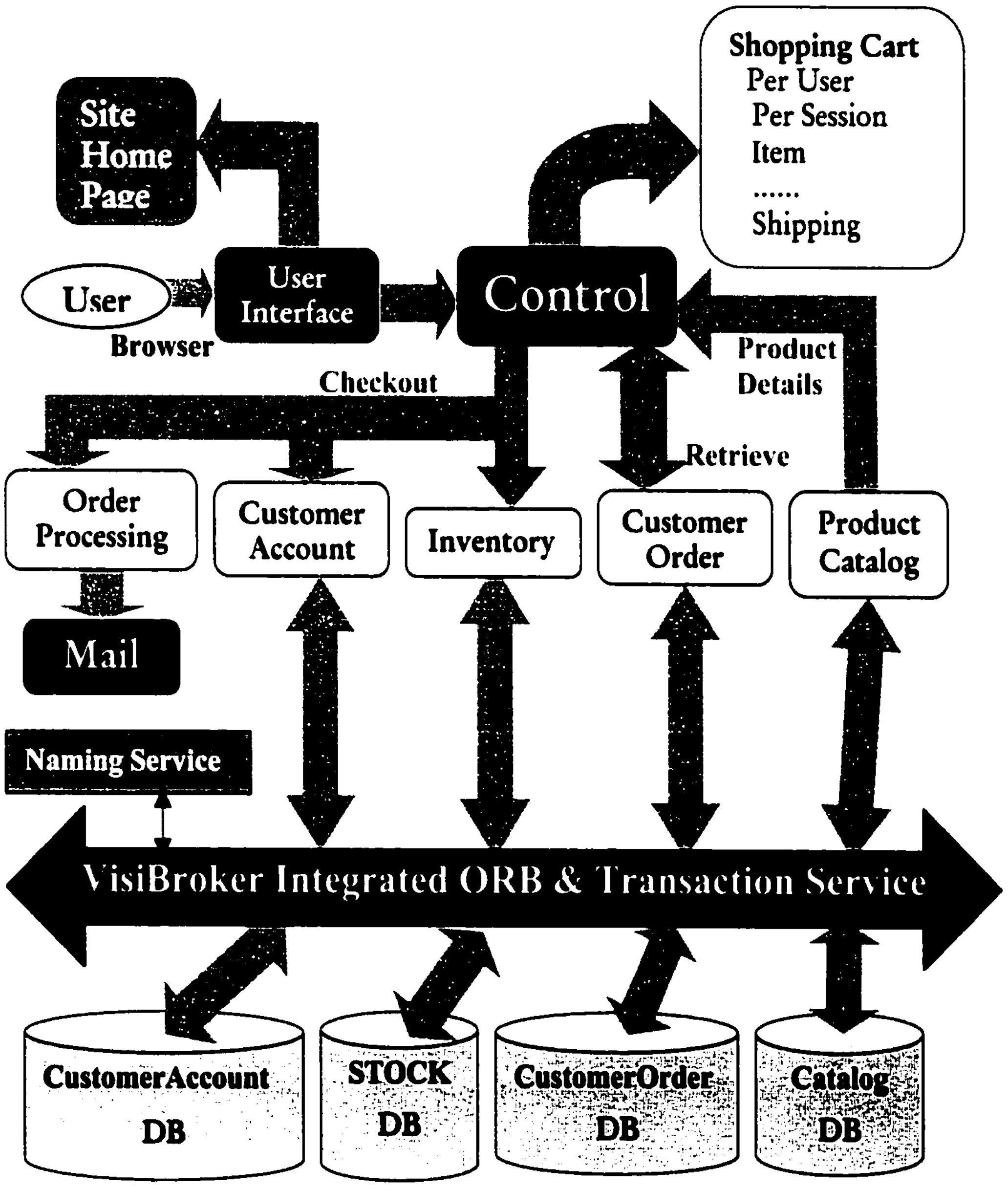


Figure 5.3: Functional Modules for the Prototype

5.3.4 MVC Architecture

Because our system primarily focuses on large scale, complex applications, so we need framework during development. To provide robust and scalable software, our system has based its solution design around some universally accepted design techniques. Our viable template is based on the MVC design pattern and distributed component-based technologies. And a real application example (prototype) is developed to verify this template. We give a brief overview of MVC design pattern and then describe how our system follows the MVC design pattern.

5.3.4.1 MVC Design Pattern

The Model/View/Controller (MVC) architecture is a well-known design for GUI objects that dates back to the SmallTalk language [KP88]. Before MVC user interface designs tended to lump these objects together. MVC [GHJV95] decouples them to increase flexibility and reusability. MVC consists of three kinds of objects: a Model, a View and a Controller.

- The Model – the elements of logic that actually process the application business logic. For example, the logic that does a database update to add an item to a user's shopping cart.
- The View – the elements of logic that construct the static and dynamic HTML pages sent to user, thus determining the presentation form and style of the results of the interaction.
- The Controller – the elements of logic that defines the way the user interface reacts to user input.

By making the view completely independent of the controller and model, front-end clients can easily be substituted. Also, keeping controller and model code out of the view, persons who not understand this code cannot change things they should not change. Keeping the controller and model separate lets you change the controller without interfering with the model and change the model without interfering with the controller. The MVC framework supports the use of different program components, with well-defined interfaces between them, for each of the different types of logic. The major benefits of using MVC design pattern is that we can split our application into three fairly distinct sections, each requiring different skills, allows us to better manage our development cycle and team.

This thesis design follows the MVC design pattern: The JSPs (and static HTML pages) provides the view, some servlets and JSPs are the controller and the JavaBeans, CORBA Objects acts as the role of the model.

5.3.4.2 The View

The view component of the sample application is responsible for generating the static and dynamic HTML pages that will be returned to the client. In our sample application, the implementation of the view is contained completely in the Web tier and three kinds of component work together to implement the view: JSP pages, custom JSP actions, and JavaBean components.

JSP pages are used for dynamic generation of HTML responses. Custom JSP actions make it easier for JSP pages to use JavaBeans components, which is the contract between JSP pages and the model. The JSP pages rely on these JavaBeans and CORBA objects to read model data to be rendered to HTML, while elsewhere in the system, the model and controller coordinate to keep the JavaBeans components up to date.

The user interface for the shopping interaction consists of a set of screens, which are the total content delivered to the browser when the user requests an application URL. The following list identifies what model information it presents:

- Main_Screen

This is the home page of the application. It displays a list of all book categories in the catalog. The customer can click on any category to browse through a master view of products that belong in that category.

- Category_Screen

This screen displays a view of all books that belong to this particular category. The customer can click on the name of any book on display to see the book detail and also add to his shopping cart.

- Book_Detail_Screen

This screen display detailed information about a particular book item. It also provides an AddtoCart button. Clicking this button adds the book currently being shown to the shopping cart and displays the resulting shopping cart.

- Cart_Screen

This screen displays the contents of the customer's shopping cart. For each item in the shopping cart, it includes a brief description of the item and its quantity. The customer can change the quantity of each item and remove the items in the whole shopping cart. It also has a checkout button. Clicking the checkout button initiates the process of placing an order.

- Provide_Information_Screen

This screen displays a form to let the customer fills. The BookStore will ship the ordered books to this customer according to the information and in the meantime send the confirmation email. Customer could places the order by clicking the continue button.

- Receipt_Screen

This screen displays the receipt after an order has been confirmed and committed. It shows a unique order identifier so that the customer can track the order later on. It also shows shipping and billing information.

- Retrieve_Order_Screen

This screen displays the form allowing the customer provide order information to check his order information.

5.3.4.3 The Model

The business logic part of an interaction is isolated from the details of Web technology in our system. And system also can store state in the Web tier using the state maintenance capabilities of servlets and JSP, which include the *HttpSession* and *ServletContext* objects as well as JavaBeans components. The business logic is wrapped with JavaBeans, which can be serialized and sent via a protocol (IIOP, HTTP, etc.) to a remote server to be executed and then sent back. This allows for very efficient communication.

The business logic part is the piece of code ultimately responsible for satisfying client requests. The business logic in our system address a wide range of potential requirements which include ensuring transactional integrity of application components, maintaining and accessing application data in a consistent state, and integrating new application components with existing applications. The business logic in our system provides query and update access to Database using SQL and JDBC interfaces. This includes initiating CORBA objects updating the database using flat-transaction with VisiBroker Integrated Transaction Service (ITS).

5.3.4.4 The Controller

The thesis framework supports the development of interaction controller logic using either Java Servlets or Java ServerPages (JSP) technology. Both of these implementation mechanisms have significant advantages over using CGI-BIN or Web server plugins. First of all, Servlets and JSPs execute in process in the context of the Java Virtual Machine thus providing all the benefits and facilities of the Java runtime. Secondly, by their very nature, Web applications written to the Java programming model are portable across a wide range of platforms. Applications can easily be migrated without recompile.

The Servlets and JSP pages, which act as Controller in the sample application, are responsible for coordinating the model and view. Those pages accept user gestures from the view, translate them into business events based on the behavior of the application, and process these events. The processing of an event involves invoking methods of the model to cause the desired state changes. Finally, the controller selects the screen shown in response to the request that was processed. The controller coordinates both the view and the data.

5.4 System Functionality

Our system models a typical Web transaction – an online Bookstore to verify our approach. The system interface is presented to its customers through a Web site and a customer interacts with the system using a Web browser.

Like a typical e-commerce site, the Bookstore presents the customer with a catalog of books. The customer selects items of interest and places them in a shopping cart. When the customer has selected the desired items and indicates readiness to buy what is in the shopping cart, the system displays a bill of sale: a list of selected items, a quantity for each item, the price of each item, and the total cost. The customer can revise or cancel the order. When the customer ready to accept the order, the customer provides personal information, a credit card number to cover the costs and supplies a shipping address.

5.4.1 Shopping Scenario

The shopping scenario describes a user's view of interactions with the system. It allows shoppers to buy items online.

The primary function of the shopping interaction is to provide an interface where customers can browse through and purchase items. This shopping interaction starts with the customer's visit to the application system home page and ends when the customer orders from the site:

(1) A customer connects to the system, by pointing the browser to the URL for the system's home page. This allows the customer to browse through the catalog.

(2) The customer browses through the catalog. The customer can select a category to see a list of all the books in that category. For example, the customer can select the category *Children's Books* to view all children's books that the Java Bookstore sells.

(3) The customer selects a particular book in the list. Then, the system displays detailed information about the selected book. The description of the book is shown along with pricing information.

(4) The customer decides to purchase a particular book and clicks *Add to Cart* button to add the item to the shopping cart. As the customer may continue shopping, adding more items to the shopping cart. As the customer browses through the catalog, the system remembers all the items placed in the cart. The system can recall the shopping cart at any time during the interaction to review or revise the contents of the cart.

(5) A checkout button is presented along with the shopping cart. The system present a summary of all items that would be ordered along with their costs. The customer can choose to order the items in the shopping cart at any time.

(6) When customer asks to checkout, the system will present a form for the customer entering his billing information, including name, password, shipping address, email address, and credit card number.

(7) After the customer finish the form, he confirms the order and application system accepts the order for delivery. A receipt including a unique order number and other order details is presented to the customer.

Although this scenario presents the system from a single customer's point of view, our Bookstore system needs to simultaneously support a large number of shoppers.

5.4.2 Transaction Scenario

Atomic transactions are a well-known technique for guaranteeing application consistency in the presence of failures. With the advent of Java and CORBA technology, it is possible to ensure consistency, reliability, and integrity of distributed transaction system.

The scenario for the prototype involves a Bookstore that has several data information tables. The sample application's persistent data is stored in five databases: *Catalog, CustomerAccount, CustomerOrder, STOCK, TempOrder* database table. The *Catalog* database holds information about books in this Bookstore. The *CustomerAccount* database holds information about customer information. The *CustomerOrder* database holds information about every customer order information. The *STOCK* database holds information about quantity of each book.

After the customer confirms the order and application system, System controller must access three database tables – *CustomerAccount, CustomerOrder, STOCK* table. The sample application uses VisiBroker ITS (Integrated Transaction Services) to accurately reduce the inventory of ordered products from STOCK (database) table and add a new order entry to the order (*CustomerOrder* database) table, and in the meantime add this customer information into *CustomerAccount* database table in an atomic operation.

(1) Beginning the transaction – The transaction originator (***doTransit*** program) registers with the ITS Transaction Service its desire to begin a transaction. ITS Transaction Service answers this request by returning a transaction context to the transaction originator.

(2) Issuing requests to transactional objects (*StockItem* objects and *StockStorage* objects) – The ITS Transaction Service propagates the transaction context to all objects participating in the transaction.

(3) Completing a transaction – A transaction can be completed in the following ways:
   - If a commit is requested and all participating Resources agree to commit, the changes are committed. If any participant votes for rollback, then the transaction is rolled back.
   - If completion is not requested by the application, the ITS Transaction Service will rollback the transaction when the timeout period.expires.

Our system ensures atomic transaction, recovery and logging, integration with databases and legacy systems in Web-based distributed transaction environment with VisiBroker ITS (Integrated Transaction Services).

5.5 System Modeling with UML

Modeling is a central part of all the activities that lead up to the deployment of good software. We build models to communicate the desired structure and behavior of our system; to visualize and control the system's architecture; to better understand the system we are building, often exposing opportunities for simplification and reuse; to manage risk [BRJ99].

The UML is a well-defined and widely accepted response to that need. The Unified Modeling Language (UML) is the industry-standard visual modeling language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [BRJ99] [DW98].

We explore the UML modeling technique to describe system development.

5.5.1 Shopping Scenario Modeling

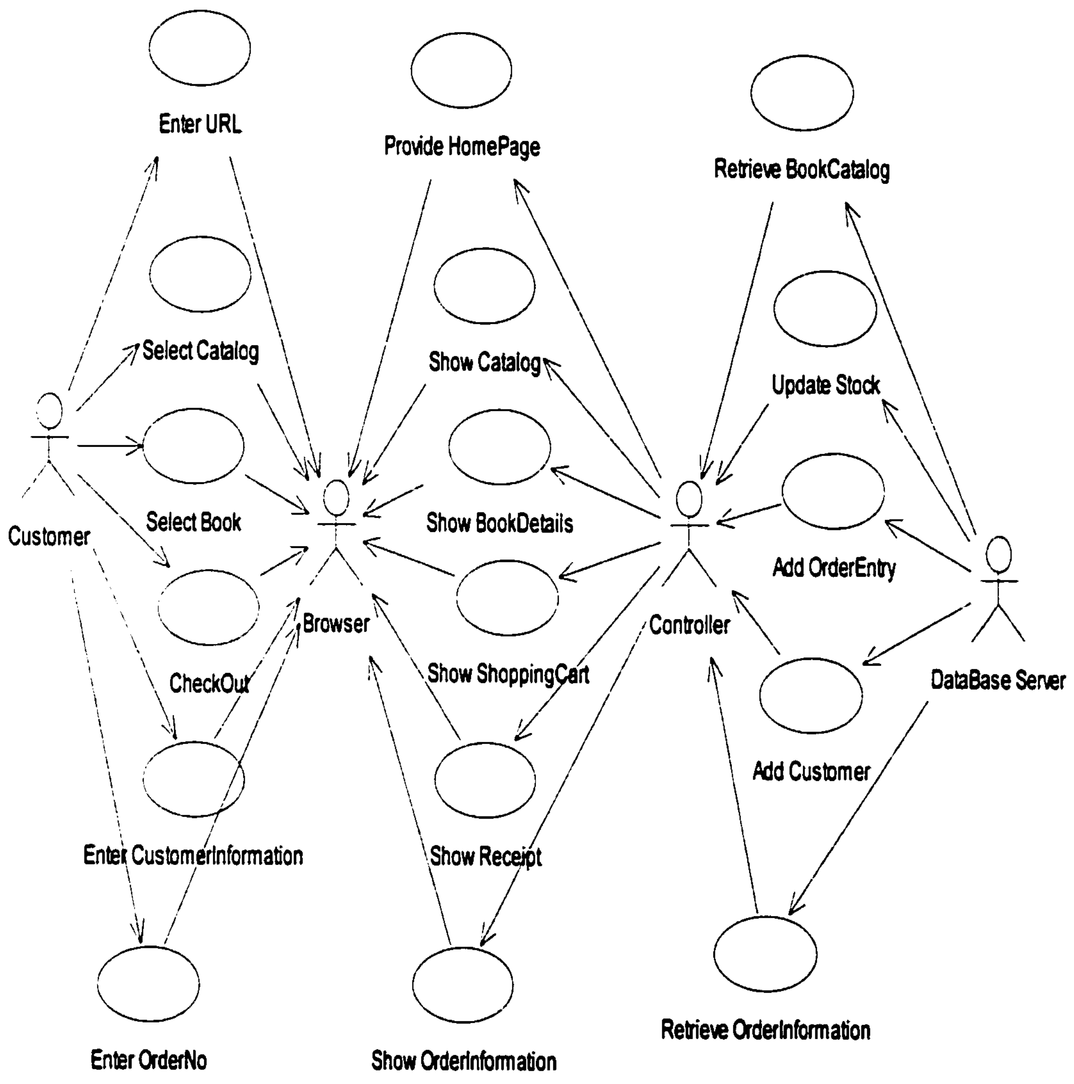
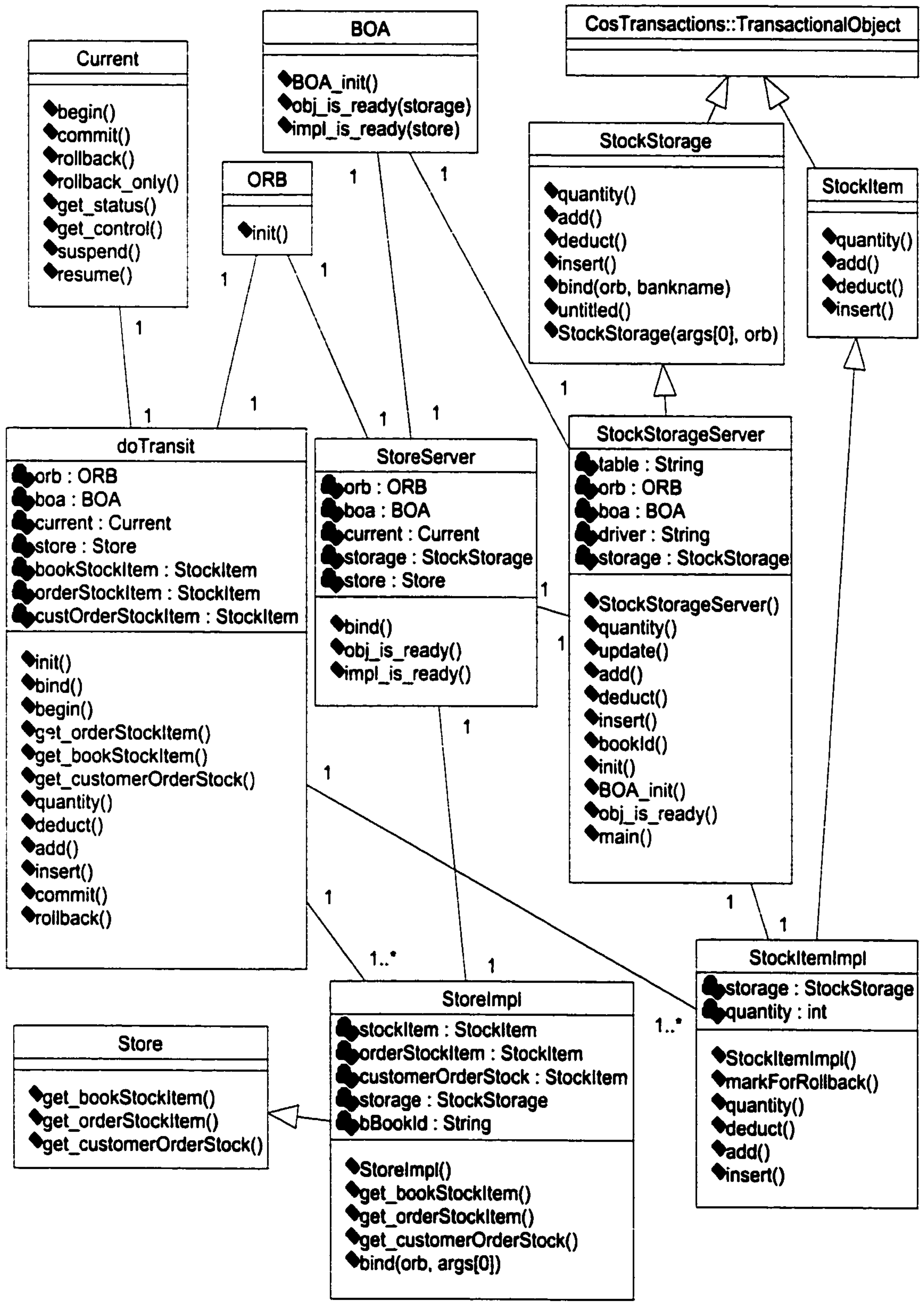


Figure 5.4 Shopping Scenario Use Case Diagram

5.5.2 Transaction Scenario Modeling

Figure 5.5: Transaction Scenario Class Diagram



CosTransactions::TransactionalObject
BOA
◆BOA_init()
◆obj_is_ready(storage)
◆impl_is_ready(store)
Current
◆begin()
◆commit()
◆rollback()
◆rollback_only()
◆get_status()
◆get_control()
◆suspend()
◆resume()
ORB
◆init()
StockStorage
◆quantity()
◆add()
◆deduct()
◆insert()
◆bind(orb, bankname)
◆untitled()
◆StockStorage(args[0], orb)
StockItem
◆quantity()
◆add()
◆deduct()
◆insert()
doTransit
orb : ORB
boa : BOA
current : Current
store : Store
bookStockItem : StockItem
orderStockItem : StockItem
custOrderStockItem : StockItem
◆init()
◆bind()
◆begin()
◆get_orderStockItem()
◆get_bookStockItem()
◆get_customerOrderStock()
◆quantity()
◆deduct()
◆add()
◆insert()
◆commit()
◆rollback()
StoreServer
orb : ORB
boa : BOA
current : Current
storage : StockStorage
store : Store
◆bind()
◆obj_is_ready()
◆impl_is_ready()
StockStorageServer
table : String
orb : ORB
boa : BOA
driver : String
storage : StockStorage
◆StockStorageServer()
◆quantity()
◆update()
◆add()
◆deduct()
◆insert()
◆bookId()
◆init()
◆BOA_init()
◆obj_is_ready()
◆main()
StockItemImpl
storage : StockStorage
quantity : int
◆StockItemImpl()
◆markForRollback()
◆quantity()
◆deduct()
◆add()
◆insert()
Store
◆get_bookStockItem()
◆get_orderStockItem()
◆get_customerOrderStock()
StoreImpl
stockItem : StockItem
orderStockItem : StockItem
customerOrderStock : StockItem
storage : StockStorage
bBookId : String
◆StoreImpl()
◆get_bookStockItem()
◆get_orderStockItem()
◆get_customerOrderStock()
◆bind(orb, args[0])
1
1..*

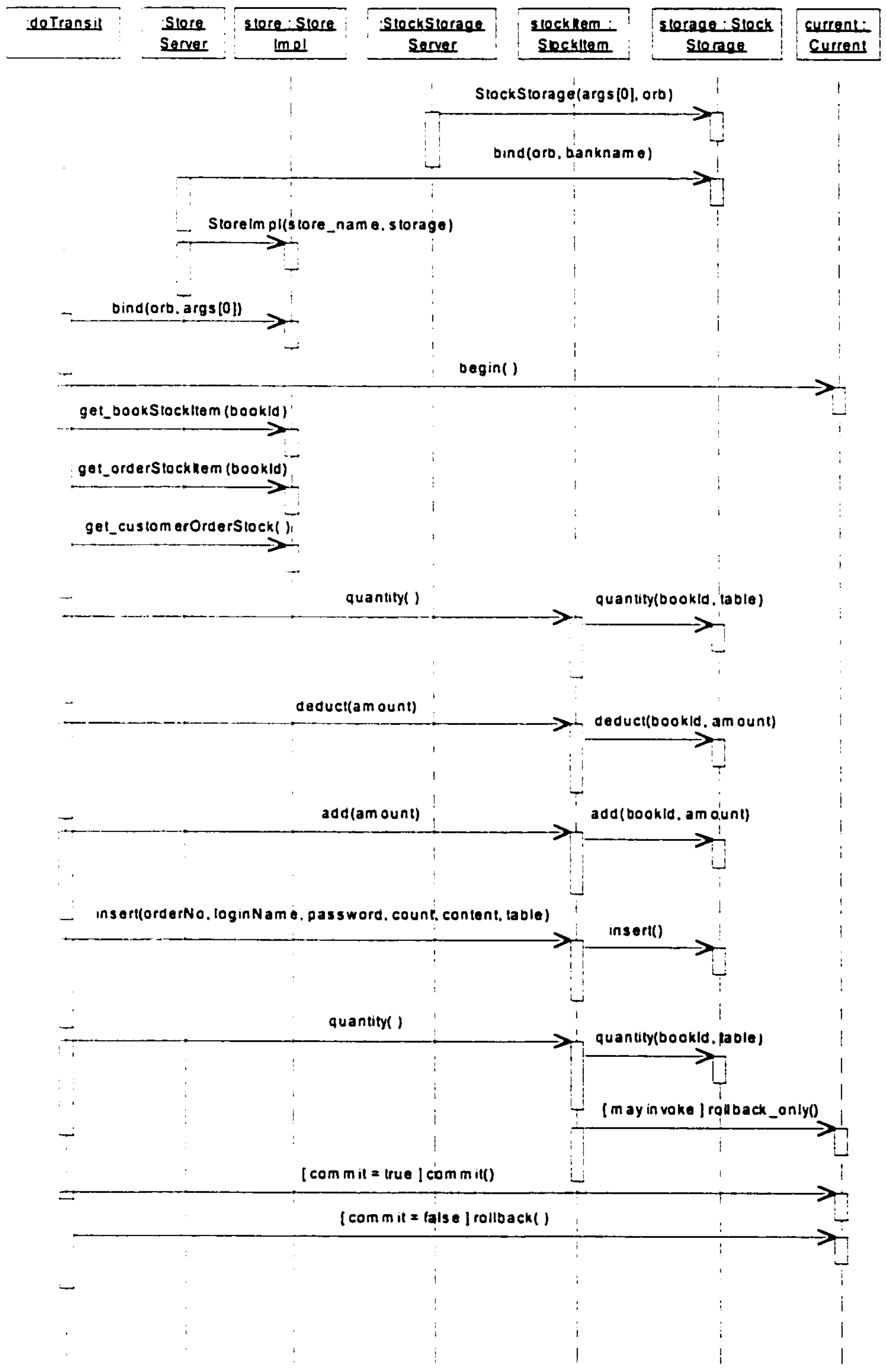


Figure 5.6: Transaction Scenario Interaction Diagram

5.6 The Contracts among Transaction Components – IDL Interface

OMG IDL provides the basis of agreement about what can be requested of an object implementation via the ORB. However, IDL is not just a guide to clients of objects. IDL compilers use interface definitions to create the means by which a client can invoke a local function and invocation on an object on another machine across network. The code generated for the client to use is stub code, and the code generated for the object implementation is called skeleton code.

OMG IDL is a declarative language for defining the interfaces of CORBA object. It is also language-independent. As IDL interface acts as a contract between developers of objects and the eventual users of their interfaces, it can also be used as a design tool for partitioning an application or system into components.

5.6.1 The IDL for ITS Transaction Service Standard

The *CosTransactions* module is in the Transaction Service IDL that conforms to the final OMG Transaction Service version 1.1. The interfaces in the *CosTransactions* module have been defined by OMG at its CORBA Service specification [OMG96]. Any object participating in the transaction must inherit from *CosTransactions::TransactionalObject.*

```
module CosTransactions
{
  exception HeuristicRollback {};
  exception HeuristicCommit {};
  exception HeuristicMixed {};
  exception HeuristicHazard {};
  exception SubtransactionsUnavailable {};
  exception NotSubtransaction {};
  exception Inactive {};
  exception NotPrepared {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};
exception SynchronizationUnavailable {};

// Current transaction
//interface Current : CORBA::ORB::Current
interface Current
{
    void begin() raises(SubtransactionsUnavailable);
    void commit(in boolean report_heuristics)
        raises(NoTransaction, HeuristicMixed, HeuristicHazard);
    void rollback() raises(NoTransaction);
    void rollback_only() raises(NoTransaction);
    Status get_status();
    string get_transaction_name();
    void set_timeout(in unsigned long seconds);
    Control get_control();
    Control suspend();
    void resume(in Control which) raises(InvalidControl);
};

interface TransactionFactory
{
    Control create(in unsigned long time_out);
    Control recreate(in PropagationContext ctx);
};

interface Control
{
    Terminator get_terminator() raises(Unavailable);
    Coordinator get_coordinator() raises(Unavailable);
};
```

```
interface Terminator
{
    void commit(in boolean report_heuristics)
        raises(HeuristicMixed, HeuristicHazard);
    void rollback();
};


interface Coordinator
{
    Status get_status();
    Status get_parent_status();
    Status get_top_level_status();
    boolean is_same_transaction(in Coordinator tc);
    boolean is_related_transaction(in Coordinator tc);
    boolean is_ancestor_transaction(in Coordinator tc);
    boolean is_descendant_transaction(in Coordinator tc);
    boolean is_top_level_transaction();
    unsigned long hash_transaction();
    unsigned long hash_top_level_tran();
    RecoveryCoordinator register_resource(in Resource r) raises(Inactive);
    void register_synchronization (in Synchronization sync)
        raises(Inactive, SynchronizationUnavailable);
    void register_subtran_aware(in SubtransactionAwareResource r)
        raises(Inactive, NotSubtransaction);
    void rollback_only() raises(Inactive);
    string get_transaction_name();
    Control create_subtransaction() raises(SubtransactionsUnavailable, Inactive);
    PropagationContext get_txcontext () raises(Unavailable);
};
```

```
interface RecoveryCoordinator
{
    Status replay_completion(in Resource r) raises(NotPrepared);
};

interface Resource
{
    Vote prepare() raises(HeuristicMixed, HeuristicHazard);
    void rollback() raises(HeuristicCommit, HeuristicMixed, HeuristicHazard);
    void commit() raises(NotPrepared, HeuristicRollback, HeuristicMixed,
                        HeuristicHazard);
    void commit_one_phase() raises(HeuristicHazard);
    void forget();
};

interface TransactionalObject{ };

interface Synchronization : TransactionalObject
{
    void before_completion();
    void after_completion(in Status status);
};

interface SubtransactionAwareResource : Resource
{
    void commit_subtransaction(in Coordinator parent);
    void rollback_subtransaction();
};

};
```

5.6.2 The IDL for ITS Transaction Service Extensions

The *VISTransactions* module is in the ITS extensions to the standard. These interfaces have been defined by OMG at its CORBA Service specification [OMG96].

```
module VISTransactions
{
 interface Current : CosTransactions::Current
 {
      voidbegin_with_name(in string userTransactionName)
           raises(CosTransactions::SubtransactionsUnavailable);

      CosTransactions::RecoveryCoordinator
           register_resource(in CosTransactions::Resource r)
                     raises(CosTransactions::Inactive);

      void register_synchronization(in CosTransactions::Synchronization s)
           raises(CosTransactions::NoTransaction, CosTransactions::Inactive,
      CosTransactions::SynchronizationUnavailable, CosTransactions::Unavailable);

      CosTransactions::otid_t get_otid()
           raises(CosTransactions::NoTransaction, CosTransactions::Unavailable);

      CosTransactions::PropagationContext get_txcontext()
                raises(CosTransactions::Unavailable, CosTransactions::NoTransaction);

      attribute string        ots_name;
      attribute string        ots_host;
      attribute string        ots_factory;
 };
 interface TransactionFactory : CosTransactions::TransactionFactory
 {
      CosTransactions::Control  create_with_name(in unsigned long time_out,
                                          in string userDefTxName);

      unsigned long up_since();
 };

};
```

5.6.3 The IDL for the Prototype

```
// myBookStock.idl
#include "CosTransactions.idl"
#pragma prefix "visigenic.com"

module myBookStore {

    interface StockItem : CosTransactions::TransactionalObject {

       long quantity();
       void add( in long amount );
       void deduct( in long amount );
       void insert( in string orderNo, in string loginName, in string password,
                    in long count, in string content, in string table );
    };

    exception NoSuchStockItem {

       string bookId;
    };
    exception NoSuchCustomerOrder { };

    interface Store {

       StockItem get_bookStockItem( in string bookId ) raises( NoSuchStockItem );
       StockItem get_orderStockItem( in string bookId ) raises( NoSuchStockItem );
       StockItem get_customerOrderStock() raises( NoSuchCustomerOrder );
    };

    interface StockStorage : CosTransactions::TransactionalObject {

        long quantity( in string bookId, in string table ) raises( NoSuchStockItem );
        void add( in string bookId, in long amount ) raises( NoSuchStockItem );
        void deduct( in string bookId, in long amount ) raises( NoSuchStockItem );
        void insert( in string orderNo, in string loginName, in string password,
            in long count, in string content, in string table )raises( NoSuchCustomerOrder
);
       string bookId( in string bookId );
    };

};
```

5.7 Detailed Transaction Implementation of Prototype

The scenario for the prototype involves a Book Store that has several data information table. After customer commit an order, the STOCK table must be updated, a new order entry must be added to the *CustomerOrder* table, and this customer information must be added to the *CustomerAccount* table during a transaction.

(i) Implement the transaction originator (client program)

The file named ***doTransit.java*** contains the implementation of the Java client program that is also the transaction originator. The ***doTransit*** program gathers the information from JSP page and performs a single ITS-managed transaction. The ***doTransit*** program performs these tasks:

- Initializes the ORB with the System Properties

```
...
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);
...
```

- Binds to the Store object
- Begins a transaction

```
...
//Start an ITS-managed transaction
Current current;
{
  org.omg.CORBA.Object initRef =
      orb.resolve_initial_references( "TransactionCurrent" );
  current = CurrentHelper.narrow( initRef );
}
// Begin the transaction
current.begin();
...
```

- Obtains a reference to the transactional (StockItem) objects

```
...
StockItem bookStockItem = store.get_bookStockItem( bookId );
StockItem orderStockItem = store.get_orderStockItem( bookId );
StockItem custOrderStockItem = store.get_customerOrderStock();
...
```

- Invokes the deduct(), add() and insert() methods on the StockItem objects

```
...
bookStockItem.deduct( quantity );
orderStockItem.add( quantity );
```

```
custOrderStockItem.insert( orderNo, loginName, password, count, content,
                                          "CustomerOrder" );
...
```

- Commits or rolls back the transaction – Once a transaction has begun, it must be committed or rolled back to complete the transaction. If an originator of an ITS-managed transaction does not complete the transaction, the ITS Transaction Service will rollback the transaction after a timeout period.

```
...
boolean commit = false;
try {
...
commit = true;
}
...
finally
{
    // Commit or rollback the transaction
    if( commit )
    {
        System.out.println( "*** Committing transaction ***" );
        current.commit( false );
    }
    else
    {
        System.out.println( "*** Rolling back transaction ***" );
        current.rollback();
    }
}
...
```

(2) Implement the ***StoreServer*** program: Initialize the ORB and BOA, obtains a StockStorage object and create a Store object with it, register the Store object with the BOA, and enters a loop prepare to receive client requests.

(3) Implement the ***Store***: Instantiate and return a transactional object (*StockItem*) upon request.

(4) Implement a transactional object (***StockItem*** object): Handle requests to view quantity, deduct and add quantity of items.

(5) Implement the ***StockStorage*** object: Access and update persistent data as requested by business (StockItem) objects in database tables.

(6) Implement the ***StockStorageServer***: Initialize the ORB and BOA, instantiate a *StockStorageServer* object, and register it with the BOA. Access the Oracle database using the ITS JDBC DirectConnect driver and Oracle JDBC driver. Implement data access from a transactional object (*StockStorageServer*)

5.8 Development Environment

5.8.1 Platform

The implementation of the Internet Transaction System is primarily carried out on a Windows NT server 4.0. This machine has Intel Pentium III processor at 733MHz and 128MB RAM at 133MHz speed and 10.2GB Hard Drive. Since the implementation language is Java, the platform is not really an issue here due to the fact of Java's WORA (Write Once Run Anywhere).

A Windows NT server 4.0 turns out to be the right choice for this implementation because of the following: the easy of use, availability of software and hardware, cost, and sophistication of operation system model.

5.8.2 Development Tools

As stated previously, the implementation language will be the Java language primarily due to its highly portability and Java is a pure Object Oriented Language (OOL), which is believed to be the most suitable language for component-based development [Brown98]. As a simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance language, Java also supports concurrent programming and automatic garbage collection. Its rich portable APIs and many free available tools make it an ideal tool for our system development.

The middleware or distributed infrastructure will be CORBA. This is due to the following facts:

- CORBA is one of middleware standard and component infrastructure for distributed objects endorsed by 800+ companies.
- CORBA provides interoperability across programming languages and operating system.
- CORBA is suitable for heterogeneous environment [Meta98].
- CORBA provide location transparency and legacy integration
- CORBA has a growing set of services and facilities available.

The most widely accepted standard for distributed transaction is the Object Transaction Service (OTS) from the OMG [OMG96]. VisiBroker ITS is compliant with CORBA Object Transaction Service (OTS) specification and the Java Transaction Service (JTS) specification. VisiBroker ITS integrate with popular databases and mainframes, and provides seamless access to multiple data sources, supporting both XA and non-XA environments.

As the official reference implementation of the servlet 2.1 and JSP 1.0 specifications, the JavaServer^{TM} Web Development Kit 1.0.1 (JSWDK) has been chosen as web server for testing servlets and JSP pages before they deployed to a full Web server that supports these technologies. It provides a development environment for Web-based HTML applications. It is free and reliable.

In the following, we will list an array of development tools that have been used in the system development. Core Java, Servlet/JSP, VisiBroker Integrated ORB and Transaction Service/CORBA are the major tools used at certain part of the development.

- ***JDK1.1.8***: Because the VisiBroker ITS in not compatible with the latest Java Version for Java 2 platform. Java Development Kit (JDK) 1.1.x is required.
- ***VisiBroker Integrated Transaction Service 1.1***. The VisiBroker ITS product and the VisiBroker Java Developer (ORB) must be installed.
- ***Web Server***: Java Server Web Development Kit (jswdk1.0.1) is installed in the NT as a small stand-alone web server to handle HTTP request and response.
- ***DBMS***: Oracle8I Enterprise edition has been installed in this NT to serve as a database server. Various information or data can be stored to and retrieved from this database. We can leverage its built-in SQL support, concurrency control, security etc.
- ***JDBC***: Database access is through JDBC. In this work JDBC is used to access Oracle8I database through CORBA middleware or Java Servlet/JSP in connection with Web Server.

5.9 The operation of the system

In order to run the sample application – on-line BookStore on the system, first we must start services that must be presented in this Internet transaction system. Then we follow the sequence of steps described in this section to run the sample application.

Oracle database instance Dell and JavaServer™ Web Development Kit 1.0.1 (JSWDK), VisiBroker Transaction service, Smart Agent are running as daemon processes to enable transactions across the network.

Before start the such Web server, The JSWDK Server should be configured, we must designate parameters like the port on which it listens, the directories in which it looks for HTML files, and so forth.

Server side:

The system should be running under JDK1.1.8 or JDK 1.1.x environment.

To ***start Coordinator***:

WinNT ..\Servlets>*java myBookServlet.TransitServer*

To ***start the Smart Agent*** (osagent)

Before we attempt to run web-based transactional applications, we must first start the Smart Agent on at least one host in our local network

WinNT ..\myBookStoreITS>*osagent -C*

To ***start the VisiBroker ITS Transaction Service***:

We must start an instance of the ITS Transaction service to enable transactions across the network.

WinNT ..\myBookStoreITS>*ots*

To ***start StockStorageServer*** program and access the Oracle database via Oracle OCI driver:

WinNT ..\myBookStoreITS>*vbj -DORACLE="" StockStorageServer MyStore scott tiger*

or run BATCH file:

WinNT ..\myBookStoreITS>*runStockStorageS .bat*

To ***start StoreServer*** program:

WinNT ..\myBookStoreITS>*vbj -DORBservices=com.visigenic.services.CosTransactions StoreServer MyStore*

or run BATCH file:

WinNT ..\myBookStoreITS>*runStoreServer .bat*

To ***start JavaServer™ Web Development Kit 1.0.1*** (JSWDK1.0.1)

Open the *jswdk-1.0* directory with Windows Explorer and double-click on *startserver118.bat*

(To stop the server, double-click on the *stopserver118.bat* file for graceful shutdown instead of simply killing the server process.)

Client side:

Open a Web browser, and enter following URL:

http://localhost:8080/path/myBookStore.html

6 Conclusions and Future Works

This chapter draws the conclusions and addresses the necessary extension of this thesis work.

6.1 Conclusions

In this thesis work, we have presented a methodology design for integrating the object-based transaction service within the Web application system design based on component-based technology to ensure the transaction integrity and data consistency. It is an extremely useful fault-tolerance technique, especially when multiple, remote, transactional resources are involved. Our modeling approach also enables the assembly of applications from reusable components that reside on the multiple tiers. This thesis work has provided an integrated environment that simplifies the development, deployment, and management of multi-tier distributed applications. To illustrate the functionality, interrelationship, and roles of different components and different tiers in our system, a sample prototype – Java bookstore running on such system is detailed presented to demonstrate the feasibility and effectiveness of our design approach and the overall system characteristics and functionality.

The major contribution of this thesis is:

- Combine the flexibility and reuse of object technology with the application services that now define customer service: performance, reliability, transaction integrity, and security.
- CWTS delivers the reliability and performance but in a completely new, standards-compliant way to satisfy the need for transaction control among distributed, objects-based Web applications.
- Provide an approach that has the ability to rapidly integrate, assemble the reusable component and easy to change as both the technology and application needs evolve.

- The System design with MVC design pattern, which separating the business logic from the user interface provides greater flexibility and better management during system development and maintenance.
- By adopting Internet standards and distributed computing technologies, it makes easy to deploy e-business application on our CWTS system, which ensure portability across a diverse server environment.

Overall, The CWTS framework simplifies web application development and deployment; supports heterogeneous client and server platforms; leverages existing skills and assets; delivers a scalable, reliable, and manageable environment.

6.2 Future Works

We remark that two other important issues, which have not be addressed in this thesis because of the limitations of available time and resources. These two issues are briefly outlined below, and are worth our further investigation in the future.

- Although our system modeling offer transactional guarantees to applications, these guarantee only limited to resources used at Web servers, and between servers; clients (browsers) are not included. First of our future investigations is that we should extend the transaction integrity to the Web browser and incorporate an OTS implementation into it, along with the necessary persistence and concurrency control services. Application specific transactional objects could then be constructed and downloaded into the browser on demand. These objects would then be able to participate directly in the application's transactions. Application can therefore obtain end-to-end transactional integrity, guaranteeing consistency in the presence of failures and concurrent accesses.

- As enterprise requires security guarantee that provide privacy, integrity, and authentication in Web application system. For example, if we would like to prevent some information such as credit card number, financial data got unauthorized access, personal IDs and passwords can not provide enough security. So in the second of our future investigations is that we should integrate SSL (Secure Socket Layer protocol) technology into our environment to create a secure, authenticated and encrypted communication channel between Web server and browser, and between different servers. SSL is the industry-standard protocol, which provides encryption to prevent a sniffer from deciphering confidential information in web-based communications [GSI]. Such that our site can communicate securely with any customer who uses Netscape Navigator, or Microsoft Internet Explorer and can offer secure transactions to our online customer.

Appendix A: List of Packages and Files in the system

We list here with the all source files for this system. With JSWDK, install_dir\webpages\WEB-INF\servlets is the standard location for servlet classes. While install_dir\webpages\jsp is the standard location for JSP files and install_dir\webpages\path is the standard location for HTML files.

All Servlet and JSP auxiliary Java files and Servlet files are residing in the directory:
C:\jswdk101\webpages\WEB-INF\servlets\myBookServlet

BookBean.java, BookStore.java, Cashier.java, Customer.java, CustomerAccount.java, Email.java, Order.java, OrderClient.java, OrderInfo.java, ShoppingCart.java, ShoppingCartItem.java, TransitServer.java, TransitClient.java, Transit.java, CTempOrder.java, doTransit.java, CatalogServlet.java

The ITS Java files and IDL files for Integrated Transaction are in the directory:
F:\nan\its_example\myBookStore

CreateStockTable.java, CreateTempOrderTable.java, CreateStockTable, CreateTempOrderTable, OrderVector.java

CosTransactions.idl, myBookStore.idl

StockItemImpl.java, StoreImpl.java, StockStorageServer.java, StoreServer.java

JSP Files and auxiliary image files are in the directory:
C:\jswdk101\webpages\jsp\myBookStore

JSP Files:

ArtPage.jsp, Children.jsp, Computer.jsp, Sport.jsp, BookDetail.jsp, CheckOrder.jsp, CheckOut.jsp, ErrorLog.jsp, Login.jsp, OrderInformation.jsp, Receipt.jsp, ShowCart.jsp

Image Files:

bookstoreback.jpeg, cart-3.gif, continue-orange.gif, credit-cards-welcome-visa.gif, l-library1.gif, mbox.gif, newkidsbooks.jpeg

HTML file and auxiliary image files are in the directory:

C:\jswdk101\webpages\path

HTML file:

myBookStore.html

Image Files:

inshop3.jpeg, back.gif, javalogo.gif

Bibliography:

[BBC98] Balbir Barn, Alan W. Brown and John Cheesman, "Methods and Tools for Component based Development", IEEE 1998

[BDH98] Manfred Broy, Anton Deimel, Juergen Henn, etc., "What characterizes a (software) component?", Software – Concepts & Tools (1998) 19: 49-56

[BRJ99] Grady Booch, James Rumbaugh, Ivar Jacobson, "The Unified Modeling language User Guide", Addison Wesley, 1999.

[Box98] Box, D., "Essential COM, Object Technology Series", Addison-Wesley, Reading, MA, 1998

[Brown96] A.W. Brown, "Component-Based Software Engineering", IEEE Computer Soc. Press, Los Alamitos CA, (1996)

[Brown98a] Alan Brown, "From Component Infrastructure to Component-Based Development", 1998
http://www.sei.cmu.edu/cbs/icse98/index.html

[Brown98b] Alan Brown, "Tool Support for Enterprise-Scale CBD – Determining your organization's future competitiveness", Component Strategies, Sept. 1998

[Cou96] George Coulouris, Jean Dollimore, Tim Kindberg, "Distributed Systems Concepts and Design", second edition

[DouU99] Desmond D'Souza, "Components in a Nutshell", JOOP, Feb. 1999.

[DW98] D'Souza, D. and A. Wills., "Objects, Components, and Frameworks with UML", Addison – Wesley, Reading, MA, 1998

[Eck95] Eckerson, Wayne W., "Three Tier Client/Server Architecture: Archiving Scalability, Performance, and Efficiency in Slient Server Applications", Open Information Systems 10, 1 (January 1995): 3(20)

[FM95] J.S. Fritzinger and M. Mueller, Sun Microsystems, "Java Security"
http://www.javasoft.com/security/whitepager.ps

[GHJV95] Erich Gamma, Richard Helm, Ralph Jognson, John Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995

[GN96] A. Garthwaite and S. Nettles, "Transactions for Java", MS-CIS-96-17, University of Pennsylvania, 1996

[GSI] "Guide To Securing Intranet and Extranet Servers" http://www.verisign.com/rsc/gd/ent/secure-ext/intro.html

[Hall20] Marty Hall, "Core Servlets and JavaServer Pages", Prentice Hall PTR, May, 2000

[Holz99] Steven Holzner, "Perl Core Language Little Black Book", The Coriolis Group, 1999.

[IAFE99] "IBM Application Framework for e-business: Structuring e-business applications" http://www-4.ibm.com/software/developer/library/structure/

[Jim93] Gray, Jim, and Reuter, Andreas, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann, 1993

[JLS97] James Gosling, bill Joy, Guy Steele, "The Java™ Language Specification", Sun Microsystems, Inc., 1997

[JTS99] Java Transaction Service Specification 1.0, December 1999 http://java.sun.com/products/jta/index.html

[KP88] Glenn E. Krasner and Stephen T. Pope, A cookbook for using the model-view-controller user interface paradigm in SmallTalk-80, Journal of Object-Oriented Programming, 1(3): 26-49, August/September, 1988

[Lin99] Peter van der Linden, "Not Just Java", Sun Microsystems, Software & Networking

[Little97] M.C. Little and S. K. Shrivastava, "Distributed Transactions in Java", Proceedings of the 7th International Workshop on High Performance Transaction Systems, September 1997, pp. 151-155

[Little98] M.C. Little and S. K. Shrivastava, "Java Transactions for the Internet", Distributed Systems Engineering, 5 (4), December 1998, pp. 156-167

[Little99] M.C. Little and S. K. Shrivastava, "Using the OTS to support web transactions"

[Lyon20] David Lyons, "Creating a JSP JavaBeans framework", Java Developers Journal, February, 2000

[Meta98] Meta Group Consulting, "CORBA vs. DCOM: Solution for the Enterprise" March (1998)
http://www.sun.com/whitepapers/CORBA-vs DCOM.pdf

[Micro97] The Component Object Model Specification, 1997.
http://www.microsoft.com/com

[Micro98w] Microsoft White Paper: Windows DCOM Architecture, 1998

[COM] the Microsoft COM and DCOM site
http://www.microsoft.com/som/

[OMG95] Object Management Group: "The Common Object Request Broker: Architecture and Specification"
http://www.omg.org (1995)

[OMG96] CORBAservice: Common Object Services Specification, December 1996.
http://www.omg.org

[OMG97] OMG white paper, "A Discussion of the Object Management architecture", 1997
http://www.omg.org/whitepaper

[OTS97] Object Transaction Service 1.1 Specification, December 1997
ftp://www.omg.org/pub/docs/formal/97-12-17.pdf

[ORA96] "Java in a Nutshell", O'Reilly and Associates, Inc., 1996.

[Plugin] Java™ Plug-in Product,
http://java.sun.com/products/plugin/index.html

[PS98] Frantisek Plasil, Michael Stal, "An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM", Software – Concepts & Tools (1998) 19: 14-28

[Rose94] Bill Rosenblatt, "Learning the Korn Shell", O'Reilly & Associates, Inc. Jan. 1994

[Siegel96] Jon Siegel: "CORBA Fundamentals and Programming"

[SS99] Jams Duncan Davidson, Danny Coward, "Java™ Servlet Specification, v2.2", Final Release, Sun Microsystems, December, 1999

[Stuart98] John Stuart, "Component-based development", Enterprise Middleware, Jan. 1998

[Subr99] A.V.B. Subrahmanyam, "Nuts and Bolts of Transaction Processing" http://www.subrahmanyam.com/articles/transactions/NutsAndBoltsOfTP.html

[Sun97j] Sun Microsystems: JavaBeans Specification 1.0, 1997 http://splash.javasoft.com/beans/-beans.100A.pdf

[SunE] Sun Enterprise JavaBeans site http://java.xun.com/products/ejb/index.html

[Szyp98a] Clemens Szyperski, "Emerging components software technologies – a strategic comparison", Software-Concepts & Tools, (1998) 19: 2-10

[Szyp98b] Clemens Szyperski, "Component Software: Beyond Object-Oriented Programming", ACM Press Books, Addison-Wesley. Harlow, UK, 1998

[TPMT97] Software Technology Review, "Transaction Processing Monitor Technology", January 1997, http://www.sei.cmu.edu/str/descriptions/tpmt.html

[Tuto98] Mary Campione, Kathy Walrath, "Java Tutorial Books", Second Edition, 1998, http://java.sun.com/docs/books/tutorial/book.html#2e

[VITS99] INPRISE white paper, "VisiBroker Integrated Transaction Service: Managing Transaction in a Distributed Environment" http://www.borland.com/visibroker/papers/managingits/

[Vogel98] Andreas Vogel and Keith Duddy: "Java™ Programming with CORBA"

Vita Auctoris

Name:	Nan Zhang
Place of Birth:	Shanghai, P.R.China
Year of Birth:	1967
Education:	Shanghai Second Medical University & Shanghai University of Science and Technology, Shanghai, China 1985-1991 B.Sc. A Joint Program in Biomedical Engineering
	University of Windsor, Windsor, Ontario, Canada 1998-2000 M.Sc. in Computer Science