TRANSACTION MANAGEMENT ON COLLABORATIVE APPLICATION SERVICES

BY

KOON-PO PAUL WONG

BACHELOR OF SCIENCE IN ENGINEERING (HONORS)
CIVIL AND ENVIRONMENTAL ENGINEERING
UNIVERSITY OF MICHIGAN, ANN ARBOR, MICHIGAN (1997)

MASTER OF SCIENCE
APPLIED MATHEMATICS
HARVARD UNIVERSITY, CAMBRIDGE, MASSACHUSETTS (1999)

ENG

SUBMITTED TO THE DEPARTMENT OF CIVIL AND ENVIRONMENTAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2000

SIGNATURE OF AUTHOR [illegible]
DEPARTMENT OF CIVIL AND ENVIRONMENTAL ENGINEERING
May 17, 2000

CERTIFIED BY [illegible]
FENIOSKY PEÑA-MORA
Associate Professor, Department of Civil and Environmental Engineering
Thesis Supervisor

APPROVED BY [illegible]
DANIELE VENEZIANO
Chairman, Departmental Committee on Graduate Studies

ENG

TRANSACTION MANAGEMENT ON COLLABORATIVE APPLICATION SERVICES

By

Koon-Po Paul Wong

Submitted to the Department of Civil Engineering
on May 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

In most organizations today, product development is made by teams of designers working collaboratively together in geographically distributed locations (e.g. an aircraft design). These designers frequently interact with each other to share highly dynamic design data. Such design environments demand a flexible and efficient transaction management system for concurrency management of the complex and highly interactive transactions.

This study presents the important features of such a transaction management framework. It forms an integral part of an ongoing collaboration project at MIT, called ieCollab (**I**ntelligent **E**lectronic **Collab**oration), which aims at addressing coordination and communication problems in collaborative application services.

ieCollab uses a collaborative ASP (**A**pplication **S**ervice **P**rovider) architecture which enables organizations to quickly deploy applications without the associated cost and burden of owning, managing or supporting the applications or underlying infrastructure. The collaborative ASP model is built upon the implementation of a distributed three-tier, thin-client system architecture using CORBA (Common Object Request Broker Architecture) and JDBC connectivity.

Our transaction management framework is based on the collaborative ASP model. Since collaborative transactions are usually of long duration and represent highly interactive modifications to a complex design, a more flexible and efficient transaction management system would be considered which enables collaborative transactions to complete a complex design without enduring long waits. Object-oriented database management systems provide a means to model, coordinate, store and manipulate these complex design as well as efficient concurrency.

This study presents a transaction management model on collaborative application services using an object-oriented database environment, namely ObjectStore PSE Pro. Key issues that have been emphasized include shared and grouped transactions among users, deadlock prevention and detection, transaction checkpointing, comparison of copying versus locking, nested transactions for better management, greater concurrency by semantics-based concurrency control, and specifying protocols to support our model.

Thesis Supervisor: Feniosky Peña-Mora
Title: Associate Professor of Information Technology and Project Management

Acknowledgments

- I would like to express my deepest gratitude to my advisor, Feniosky Peña-Mora, for his support, comments and suggestions throughout the development of my thesis. Feniosky has always inspired and motivated me. It has been a wonderful experience to have him as my advisor.

- My thanks to all my teammates working in the ieCollab project, Teresa Liu, Nhi Tan, Erik Abbott, Wassim El-Solh, Kaissar Gemayel, Hermawan Kamili, Saeyoon Kim, Steven Kyauk, Ivan Limansky, Kenward Ma, and Alan Ng. Without any of them, our project would never be able to develop.

- My special thanks to Erik Abbott who gives me a hand in developing the user manual for the ieCollab project.

- And most importantly, I would like to thank my parents and my brother for their constant support, love and encouragement. Their blessings have made this possible.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In most organizations today, product development is done by teams of designers. For instance, a large computer-aided design (CAD) project, such as aircraft design, typically involves a group of designers working collaboratively together in geographically distributed workstations to complete a complex design by closely interacting among themselves and dynamically sharing design data and information. Such design environments necessitate powerful data modeling, sharing and transaction management tools, efficient communication protocols and a flexible framework for concurrency management of highly interactive transactions.

Today's electronic tools in this field still mostly aim at the support of a single designer only, leaving the organization of collaboration to some organizational level not directly supported by the tools. Even though there exist tools to support concurrent use by several members of a working group, concurrent operations are mostly controlled by rudimentary, elementary mechanisms, i.e., by implicit or explicit "user locks".

This study presents the important features of supporting a collaborative transaction management framework. It forms an integral part of an ongoing collaboration project at MIT, called **ieCollab** (**I**ntelligent **E**lectronic **Collab**oration), which consists of a set of *object-oriented* tools aimed at addressing coordination and communication problems in collaborative application services.

1.2 Introduction to ieCollab

ieCollab is an Internet-based collaborative application service provider (ASP) for communicating information and sharing software applications in a protocol-rich Java meeting environment [Chen2000]. By helping create and manage virtual teams, ieCollab's collaborative solution will offer organizations new ways to improve communications, leverage off-site expertise, and reduce project costs and duration. ieCollab is developed by building on over six years of patent-pending MIT research on collaborative engineering projects with leading organizations from the manufacturing, construction, and defense software markets.

1.2.1 Requirements

A common problem faced by all corporations at all levels today is that of software application purchasing and maintenance. Many such organizations have turned to an ASP that allows them to quickly deploy applications without the associated cost and burden of owning, managing or supporting the applications or underlying infrastructure. However, typical applications deployed by an ASP do not allow for collaborative data manipulation, falling short of the needs for distributed teams to dynamically share ideas, models, simulations, calculations and information via such applications as a result of the distributed personnel. ieCollab's Internet-based meeting tools solve the problem of personnel relocation with reliable forums for communications among geographically distributed teams. The unique collaborative ASP model deployed by ieCollab will solve the requirement for an ASP with the need for collaborative software.

1.2.2 System Architecture

The ieCollab system is built upon distributed objects on a multi-tiered architecture. It comprises of three distributed layers: user services (front end user interface), business services (service logic), and data services (back end database). The first layer is linked to the second layer through CORBA (Common Object Request Broker Architecture) connection and the second layer to database via JDBC connectivity. Figure 1-1 shows the various layers and connections of the ieCollab system.

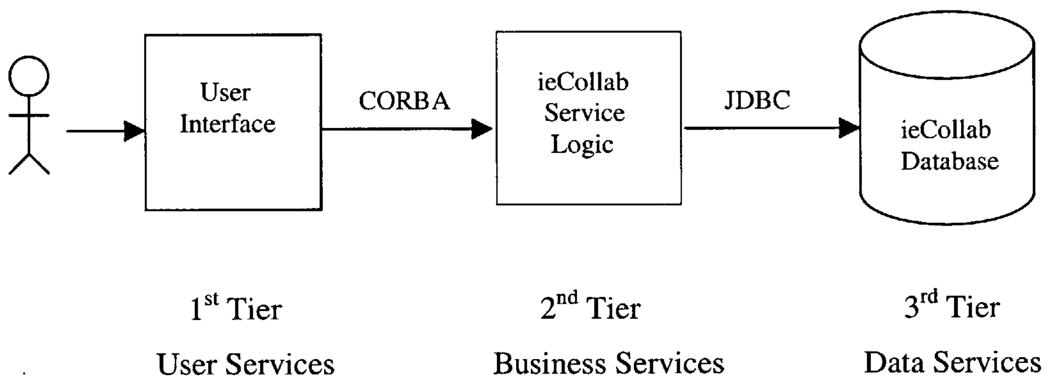


Figure 1-1: Layers and Connections of the System

User-Interface Tier:

The user-interface layer is the layer of user interaction. Its focus is on efficient user interface design and accessibility throughout an organization. The user interface layer can reside on the user's desktop, on an organization's intranet, or on the World Wide Web (Internet). The user-interface tier invokes methods on the business logic tier and thus acts as a client of the business logic servers.

Service (Server) Tier:

The service, or business logic layer, is server-based code with which the client code interacts. The business logic layer is made up of business objects – CORBA objects that perform logical business functions such as transaction management, service usage of users, frequency of access, and billing. These objects invoke methods on database tier objects.

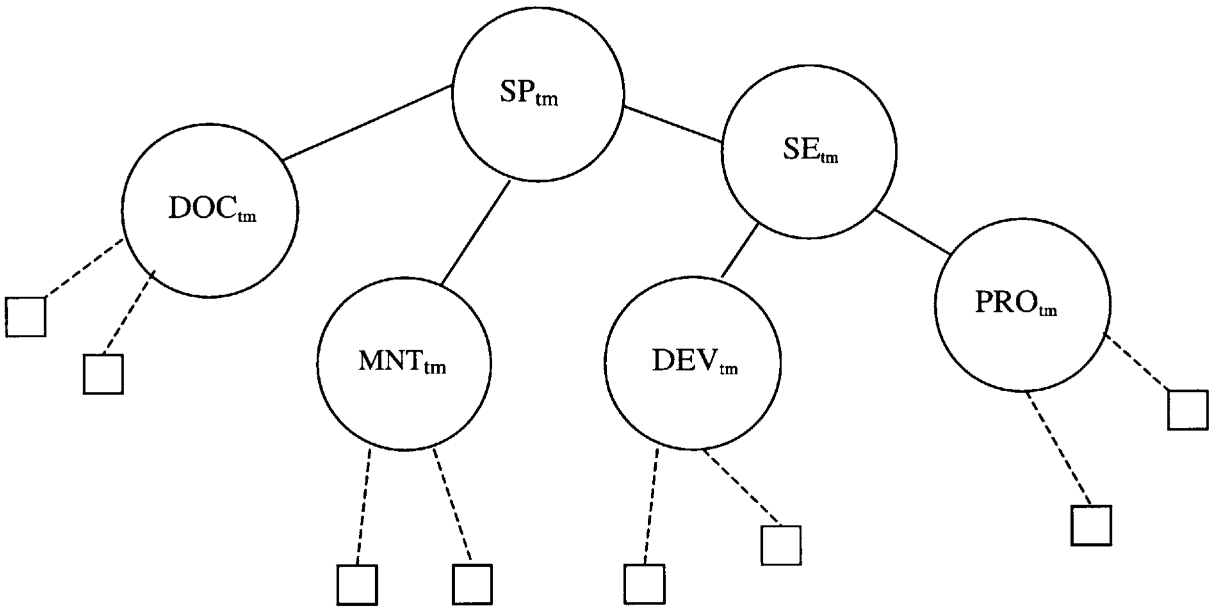
Database Tier:

The database tier is made up of objects that encapsulate database routines and interact directly with the DBMS product. All the database operations are done by individual methods. Each method that needs information from database opens connection to the database and does appropriate queries to retrieve data. After the database operation is done, connection to database would be closed.

1.3 An Example of Collaborative Product Development

Most large engineering projects, such as aircraft design, design of buildings, computer-aided software engineering (CASE), or computer-aided manufacturing (CAM) are highly collaborative in nature, i.e., they involve several participants in frequent interaction for the achievement of a common goal. These collaborative projects require a flexible and efficient transaction management system to manage the complex and highly interactive transactions.

We will introduce our proposed framework for transaction management in ieCollab with a simple example. As an illustration of collaborative product development, consider the example of software development. Imagine a Software Product organization which is divided into three divisions: *Documentation, Software Engineering*, and *Maintenance.* Software Engineering has two sub-divisions: *Development* and *Prototyping* as shown in Figure 1-2.



SP_{tm}	Transaction manager for whole Software Product organization (mediates requests from Documentation, Software Engineering and Maintenance groups)
SE_{tm}	Transaction manager for whole Software Engineering (mediates requests from Development and Prototyping)
DEV_{tm}	Transaction manager for whole Development (mediates requests from individual developers)

Figure 1-2: A Hierarchy of Transaction Managers

We will give a narrative description of the desired transaction management. This narrative maps directly to easily-implemented protocols for each of the groups.

In our example organization, Software Engineering releases documents to Documentation and code modules to Maintenance. If Software Engineering subsequently modifies a document, it must notify Documentation. Software Engineering cannot modify a code module once it has been released to Maintenance. These policies are expressed in the top-level protocol (for the whole organization), which includes the following rules:

- At the end of an operation by Software Engineering to modify a document *d*, email Documentation that *d* has been modified.
- After the end of an operation by Software Engineering to release a module *m*, reject any subsequent operation to modify *m* by Software Engineering.

Note that the first rule triggers a new request (email), and the second rule references the history of prior operations to see if it contains the event "release operation ends normally".

Within Software Engineering, Prototyping provides the Development group with prototyped versions of many of the software modules which are to be produced by Development and notifies Development when it has modified one of these versions. This policy is expressed in the protocol of the Software Engineering group, which includes the following rule:

- After the end of an operation by Prototyping to make available a module *m*, at the end of each operation by a member of the Prototyping group that modifies *m*, notify the Development group that *m* has been modified.

This rule checks a condition involving multiple events in the history of prior operations.

Within Prototyping everyone shares all objects; everyone collaborates on all aspects of the prototypes, and anyone can test or release prototyped versions. No rules are required to specify this policy. However, within Development, each designer or implementer requires exclusive use of any object he or she is modifying. In addition, if anyone changes the name of a module, the name must be changed in all programs that use that module. These policies are expressed in the protocol of the Development group, which might include the following rules:

- Between the beginning and the end of any operation by any member *D* of Development to modify a module *m*, reject any operation on module *m* by any other member of Development.
- Between the beginning and the end of any operation by any member *D'* of Development on module *m*, reject any operation by any other member of Development that modifies *m*.
- At the end of the rename operation to rename a module *m*, trigger a request to edit all modules that use *m* to modify their calls to *m*.

This example demonstrates the need for considerable collaboration work in a software development process. It highlights the rules necessary to be followed in order to achieve the desired transactions. What are the underlying transaction features required to supporting the above framework? These are enumerated in the next section.

1.4 Requirements of Transaction Management Framework in ieCollab

As illustrated in the previous example, a collaborative software development demands a powerful and flexible transaction management framework to coordinate the concurrent activities of multiple users at a time. Some of the principal requirements of the ieCollab transaction management framework are enumerated below.

- Flexible and efficient data sharing capabilities in the forms of grouped and shared transactions
- Nested transactions framework for task discretization
- Copying versus locking schemes
- Deadlock Prevention and Detection
- Effective protocols to handle conflicts or triggers among users
- Transaction checkpointing
- Semantic-based concurrency control to increase the level of concurrency

1.5 Outline of this Thesis

This chapter has introduced the primary objectives and scope of this study, which is to develop a transaction management environment for supporting collaborative application services as proposed in the ieCollab framework. The remainder of the thesis is organized as follows:

- Chapter 2 provides the basic concepts of Application Service Provider (ASP) and its thin-client system architecture for collaborative framework. It presents an overview of CORBA technology, explains the requirements of CORBA to supporting our collaborative ASP model and the benefits of implementing ASP model for such work.

- Chapter 3 outlines the concepts of JDBC connectivity which is used to access the database layer from the middle-tier service logic. With a brief discussion of the JDBC interfaces and its architecture, we examine the advantages of implementing JDBC technology for our collaborative work.

- Chapter 4 introduces the characteristics of OODBMS, and their requirements for supporting collaborative application services. By making comparison of OODBMS and RDBMS, we address the advantages of using OODBMS in the transaction management framework.

- Chapter 5 presents our transaction management model for collaborative application services, and discusses some advanced transaction management functionalities required to support the framework.

- Chapter 6 provides implementation issues of the database layer using Java-enabled ObjectStore PSE Pro.

- Chapter 7 summarizes the project results and outlines an ongoing and future work.

Chapter 2

Collaborative Application Service Provider (ASP)

Traditional enterprise applications are mostly self-contained monolithic programs which have limited access to one another's procedures and data [ASP2000]. They are usually cumbersome to build and expensive to maintain because some simple changes may require the entire program to be recompiled and retested. These create a lot of hassles of procuring hardware and installing software that a large software application requires.

With business operations now increasingly dependent on complex software applications, IT managers are embracing a new approach – web-based ASP (Application Service Provider) service - to ensuring cost-cut, secure, and business-critical applications.

2.1 What is an ASP?

Application Service Providers (ASPs) deliver and manage applications and computer services from remote data centers to multiple users via the Internet or a private network

[ASP2000]. The applications delivered over networks on a subscription basis. This delivery model speeds implementation, minimizes the expenses and risks incurred across the application life cycle, and overcomes chronic shortage of qualified technical personnel available in-house.

Large companies may decide to enlist an ASP to quickly deploy critical, enterprise applications at an affordable cost, since the resource requirements for supporting these systems have grown exponentially. Small and mid-sized organizations can deploy enterprise applications that without an ASP would involve massive investments in software, deployment time and IT personnel. These businesses can then benefit from the efficiencies of integrated, enterprise applications that were previously not cost-effective to develop and use.

2.2 Thin-Client System Architecture

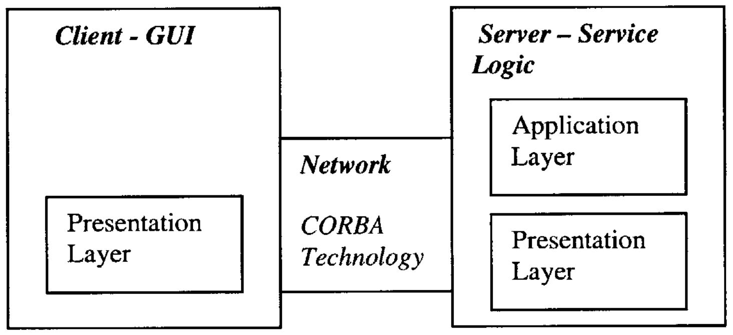


Figure 2-1: Distributed Presentation – Thin-Client System Architecture

For our ASP model, we deploy a collaborative software application. Our collaborative ASP system is built upon a distributed thin-client system architecture as shown in Figure 2-1. It composes of a presentation layer on the client side, and both application and presentation layers on the server side. The client-server system is connected with the use of CORBA technology. Since the client only implements the presentation layer, all processing is done on the server side. This creates some advantages to the system: (1) it is easy to port client to different architectures; (2) the client is decoupled from any changes in the application.

2.3 Introduction to CORBA

CORBA (Common Object Request Broker Architecture) is a popular distributed object model. It is emerged as a standard to simplify network programming and to realize component-based software architecture.

The core of the CORBA architecture is the Object Request Broker (ORB) that acts as the object bus over which objects transparently interact with other objects located locally or remotely [Chung1998]. A CORBA object is represented to the outside world by an interface with a set of methods. A particular instance of an object is identified by an object reference. The client of a CORBA object acquires its object reference and uses it as a handle to make method calls, as if the object is located in the client's address space. The ORB is responsible for all the mechanisms required to find the object's implementation, prepare it to receive the request, communicate the request to it, and carry the reply (if any) back to the client. The object implementation interacts with the ORB through either an Object Adapter (OA) or through the ORB interface.

2.3.1 System Architecture

CORBA is a distributed object framework which provides client-server type of communications. To request a service, a client invokes a method implemented by a remote object, which acts as the server in the client-server model. The service provided by the server is encapsulated as an object and the interface of an object is described in an Interface Definition Language (IDL). The interfaces defined in an IDL file serve as a contract between a server and its clients. Clients interact with a server by invoking methods described in the IDL. The actual object implementation is hidden from the client. Some object-oriented programming features are present at the IDL level, such as data encapsulation, polymorphism and single inheritance. CORBA also supports multiple inheritance at the IDL level, and CORBA IDL can also specify exceptions.

In CORBA, the interactions between a client process and an object server are implemented as object-oriented RPC-style communications. Figure 2-2 shows a typical RPC structure. To invoke a remote function, the client makes a call to the *client stub.* The stub packs the call parameters into a request message, and invokes a wire protocol to ship the message to the server. At the server side, the wire protocol delivers the message to the *server stub*, which then unpacks the request message and calls the actual function on the object.

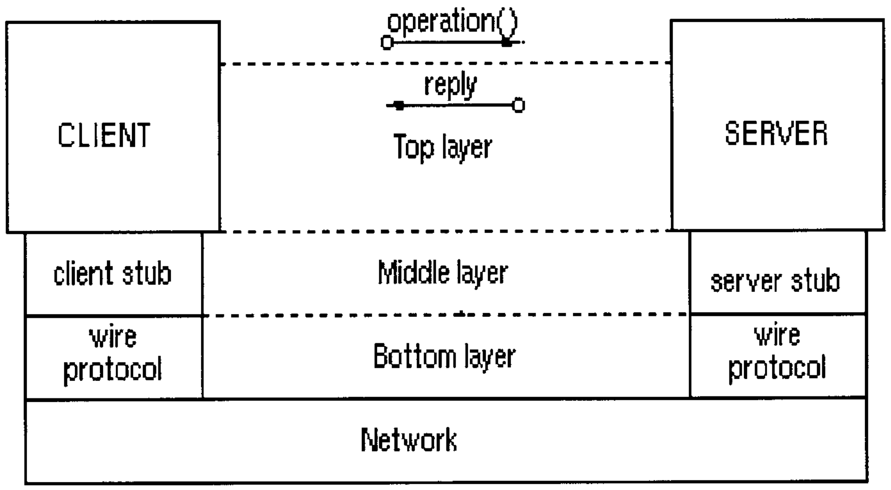


Figure 2-2: RPC structure [Chung1998]

The overall architecture of CORBA is illustrated in Figure 2-3. The top layer is the basic programming architecture, which is visible to the developers of the client and object server programs. The middle layer is the remoting architecture, which transparently makes the interface pointers or object references meaningful across different processes. The bottom layer is the wire protocol architecture, which further extends the remoting architecture to work across different machines, with TCP socket as the *de facto* standard.

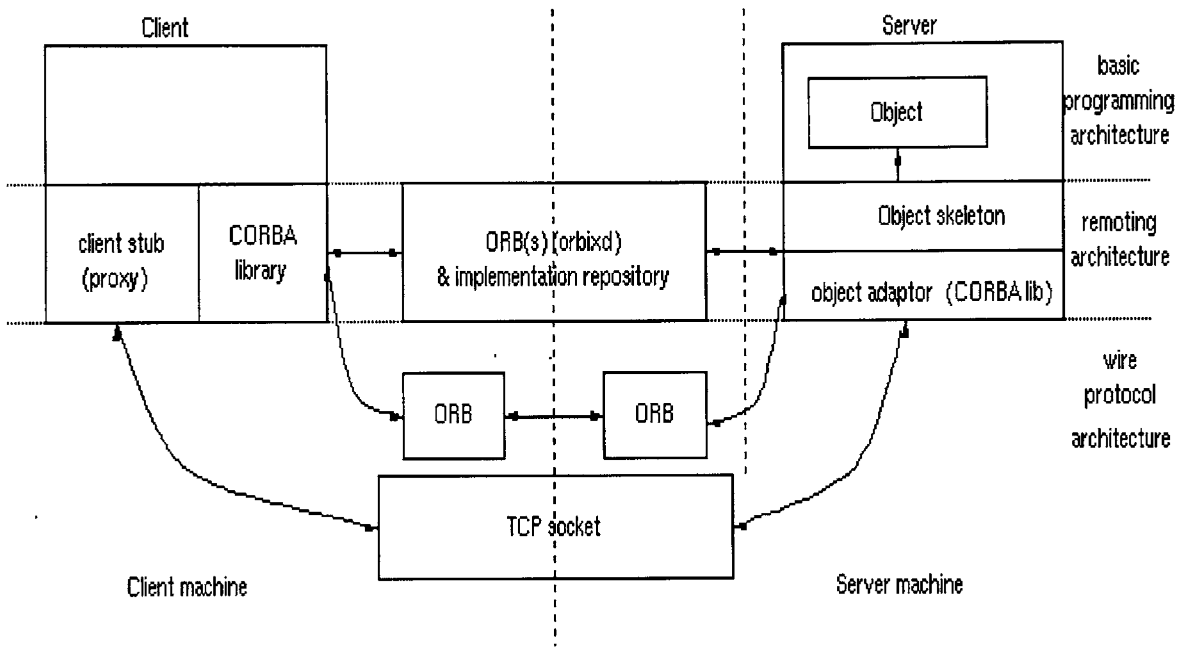


Figure 2-3: CORBA Architecture [Chung1998]

As illustrated in the Figure above, the core of the CORBA architecture is the ORB that acts as a channel connecting objects between the client and the server. Since our ASP system is built upon thin-client system architecture, ORB plays an essential role for accessing the remote application services on the server from client machines. In essence, adopting CORBA architecture is vital to support our collaborative ASP model. In the following section, we will provide the benefits of implementing ASP model for collaborative framework.

2.4 Benefits of Implementing ASP Model for Collaborative Work

There are several distinctive benefits of implementing ASP model for supporting our collaborative framework in ieCollab. These are enumerated below.

- *Operational freedom:* by outsourcing application management, we can focus critical resources on our core business function.
- *Improved performance:* ASPs can apply vast experience to implement best IT practices for superior levels of availability, security, backup, disaster recovery, and help desk (shadowing).
- *Speed to market:* the ASP already has the equipment, applications and expertise ready to provide rapid market access.
- *Financial flexibility:* ASP model reduces fixed costs and lowers overall expenditures for hardware, applications and management.
- *Reduced risk:* with no capital expenditure on software, hardware, and IT personnel, we can "test" a new technology with minimal impact to our existing environment and bottom line.

2.5 Summary

In this chapter we have presented the concepts of Application Service Provider (ASP) service and its thin-client system architecture. We also have introduced CORBA technology and overviewed its system architecture. This explains the need of CORBA for supporting collaborative application services. Finally, we have examined the benefits of implementing collaborative ASP model for our transaction management framework.

The next chapter outlines the concepts of JDBC connectivity which is used to access the database layer from the middle-tier service logic. With a discussion of the JDBC interfaces and its architecture, we examine the advantages of implementing JDBC technology for our collaborative work.

Chapter 3

Database Connectivity

3.1 Introduction

Earlier in Chapter 1, we have described the layout of the ieCollab system which composes of a three-tier system architecture. In order to access the database layer from the middle-tier application server (server logic), database connectivity has to be established. In this chapter, we will discuss the database connectivity with the use of JDBC (Java Database Connectivity).

3.2 JDBC Connectivity

With the introduction of optimizing compilers translating Java byte codes into efficient machine-specific code, the middle-tier service logic of the ieCollab system may practically be implemented by Java. Java, being robust, secure, easy to use, easy to understand, platform independent and automatically downloadable on a network, is an excellent language basis for database applications. What is needed is a way for Java applications to talk to a variety of different databases. JDBC is the mechanism for doing this.

JDBC is a Java API (Application Programming Interface) for executing SQL statements [Sun2000]. It consists of a set of classes and interfaces written in the Java programming language. Using JDBC, it is easy to send SQL statements to virtually any database. One can write a single program using the JDBC API, and the program will be able to send SQL statements to the appropriate database.

Simply put, JDBC makes it possible to do three things:

1 Establish a connection with a database

2 Send SQL statements

3 Process the results

The following code fragment gives a basic example of these three steps:

```
Connection con = DriverManager.getConnection ("jdbc:odbc:wombat", "login",
"password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
        int x = rs.getInt("a");
```

```
            String s = rs.getString("b");

            float f = rs.getFloat("c");

    }
```

3.3 JDBC Interfaces

As illustrated in the example from the previous section, the important relationships between the interfaces are as shown in Figure 3-1 (with arrows showing functions and lines showing other methods):

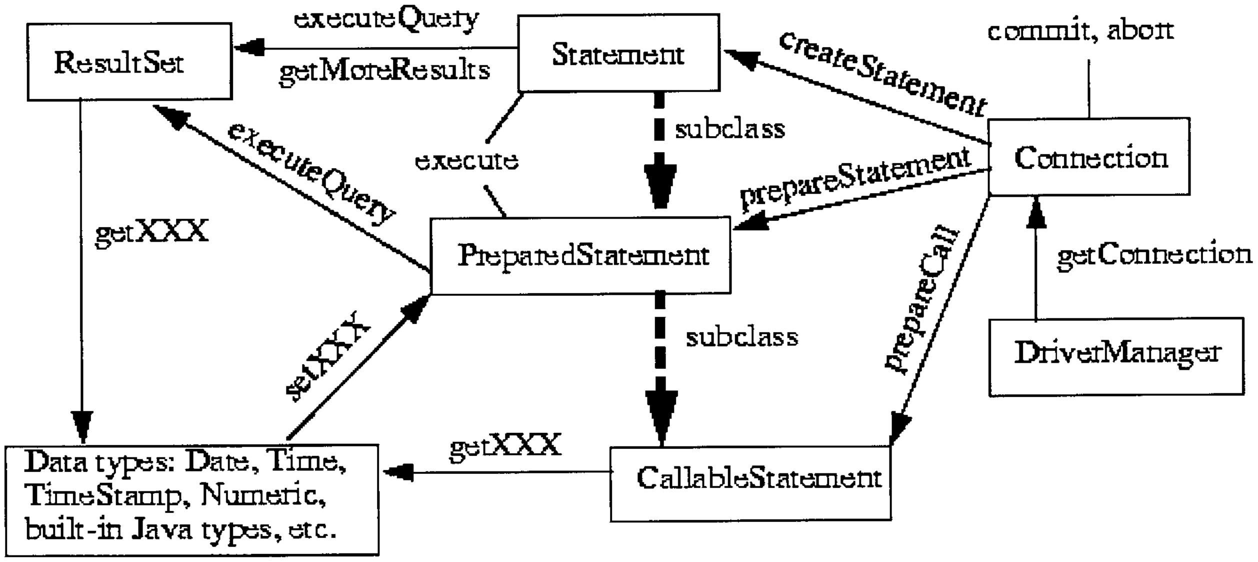


Figure 3-1: JDBC interfaces [Sun2000]

The JDBC API provides three interfaces for sending SQL statements to the database, and corresponding methods in the Connection interface create instances of them. The interfaces for sending SQL statements and the Connection methods that create them are as follows:

1. *Statement* - created by the Connection.createStatement methods. A Statement object is used for sending SQL statements with no parameters.
2. *PreparedStatement* - created by the Connection.prepareStatement methods. A PreparedStatement object is used for precompiled SQL statements. These can take one or more parameters as input arguments (IN parameters). PreparedStatement has a group of methods that set the value of IN parameters, which are sent to the database when the statement is executed. PreparedStatement extends Statement and therefore includes Statement methods. A PreparedStatement object has the potential to be more efficient than a Statement object because it has been precompiled and stored for future use. Therefore, in order to improve performance, a PreparedStatement object is sometimes used for an SQL statement that is executed many times.
3. *CallableStatement* - created by the Connection.prepareCall methods. CallableStatement objects are used to execute SQL stored procedures-a group of SQL statements that is called by name, much like invoking a function. A CallableStatement object inherits methods for handling IN parameters from PreparedStatement; it adds methods for handling OUT and INOUT parameters.

The following list gives a quick way to determine which Connection method is appropriate for creating different types of SQL statements:

- createStatement methods - for a simple SQL statement (no parameters)
- prepareStatement methods – for an SQL statement that is executed frequently
- prepareCall methods - for a call to a stored procedure

JDBC API deals with SQL conformance by allowing any query string to be passed through to an underlying DBMS driver. This means that an application is free to use as much SQL functionality as desired, but it runs the risk of receiving an error on some DBMSs.

For complex applications, JDBC deals with SQL conformance in another way. It provides descriptive information about the DBMS by means of the DatabaseMetaData interface so that applications can adapt to the requirements and capabilities of each DBMS.

3.4 JDBC Architecture and its Advantages for Collaborative Application Services

Figure 3-2 illustrates JDBC connectivity using existing database client libraries:

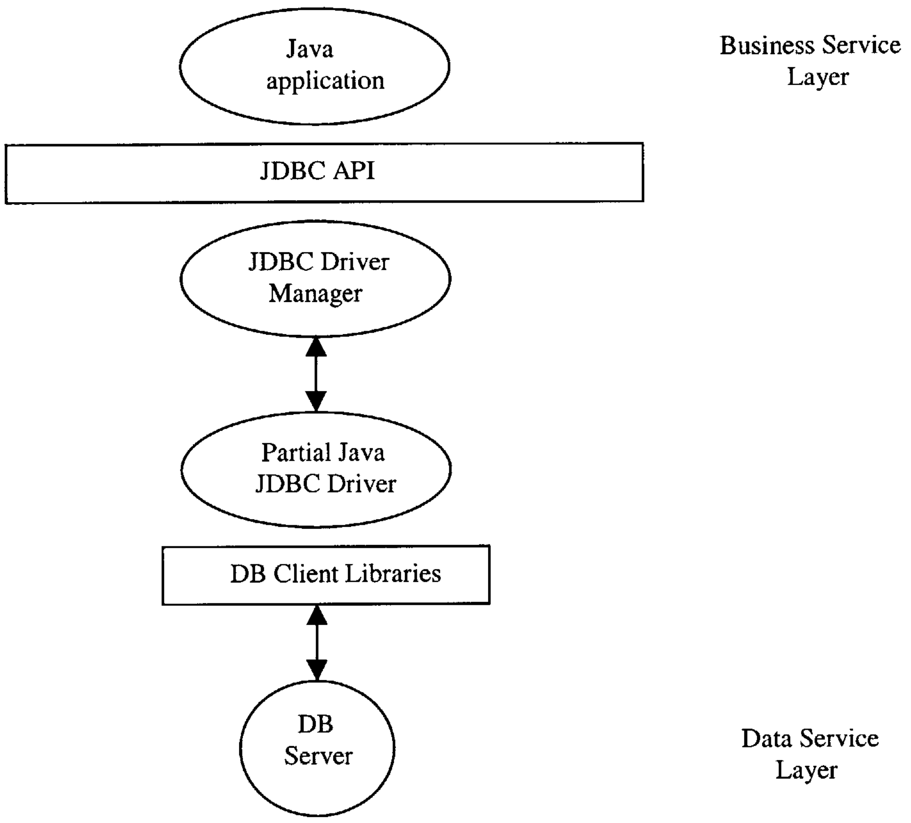


Figure 3-2: JDBC Driver and Connectivity

Using JDBC technology in database connectivity, it creates several distinctive advantages for our framework as enumerated below:

1. Simplified Enterprise Development

The combination of the Java API and the JDBC API makes application development easy and economical. JDBC hides the complexity of many data access tasks, doing most of the "heavy lifting" for the programmer behind the scenes. Hence, it facilitates the development of comprehensive programs for our distributed applications.

2. Zero Configuration for Network Computers

With the JDBC API, no configuration is required on the client side. This is essential for our thin-client system architecture as mentioned in the previous chapter. With a driver written in the Java programming language, all the information needed to make a connection is completely defined by the JDBC URL. Zero configuration for clients supports the network computing paradigm and centralizes software maintenance.

3. Database Connection Identified by URL

JDBC technology exploits the advantages of Internet-standard URLs to identify database connections. JDBC API identifies and connects to a data source, using a DataSource object, that makes code portable and easy to maintain. As such, it can support data for our geographically distributed applications across the Internet. In addition to this important advantage, DataSource objects can provide connection pooling and distributed transactions, essential for our collaborative framework.

JDBC technology generates substantial advantages in supporting our collaborative transaction management framework as discussed above. JDBC connectivity makes it possible to access the database layer from the application program (server). It is splendid for database connectivity in ieCollab.

3.5 Summary

In this chapter we have discussed JDBC connectivity as a means to access the database layer from the middle-tier server logic. We have also presented JDBC interfaces and its architecture, and finally, provided the advantages of JDBC technology for supporting transaction management framework.

The next chapter introduces the concepts of OODBMS which serves as the DBMS for our framework. By making a detailed comparison between OODBMS and RDBMS, we examine the advantages of using OODBMS to supporting collaborative application services.

Chapter 4

Object-Oriented Database Management Systems (OODBMS)

4.1 Introduction – OODBMS

An increased emphasis on process integration is a driving force for the adoption of object-oriented database systems [Ishkawa1993]. For example, the Computer Integrated Manufacturing (CIM) area is focusing heavily on using object-oriented database technology as the process integration framework. Advanced office automation systems use object-oriented database systems to handle hypermedia data. Hospital patient-care tracking systems use object-oriented database technologies for ease of use. All of these applications are characterized by having to manage complex, highly interrelated information, which is a strength of object-oriented database systems.

An initial area of focus of object-oriented database technology has been the Computer Aided Design (CAD), Computer Aided Manufacturing (CAM) and Computer Aided

Software Engineering (CASE) applications. A primary characteristic of these applications is the need to manage very complex information efficiently [Brown1991]. For example, the manufacture of an aircraft requires the tracking of millions of interdependent parts that may be assembled in different configurations. Object-oriented database management systems (OODBMS) hold the promise of putting solutions to these complex problems within reach of users.

4.1.1 Characteristics of OODBMS

Object-oriented database technology is a marriage of object-oriented programming and database technologies. Figure 4-1 illustrates how these programming and database concepts have come together to provide what we now call object-oriented databases.

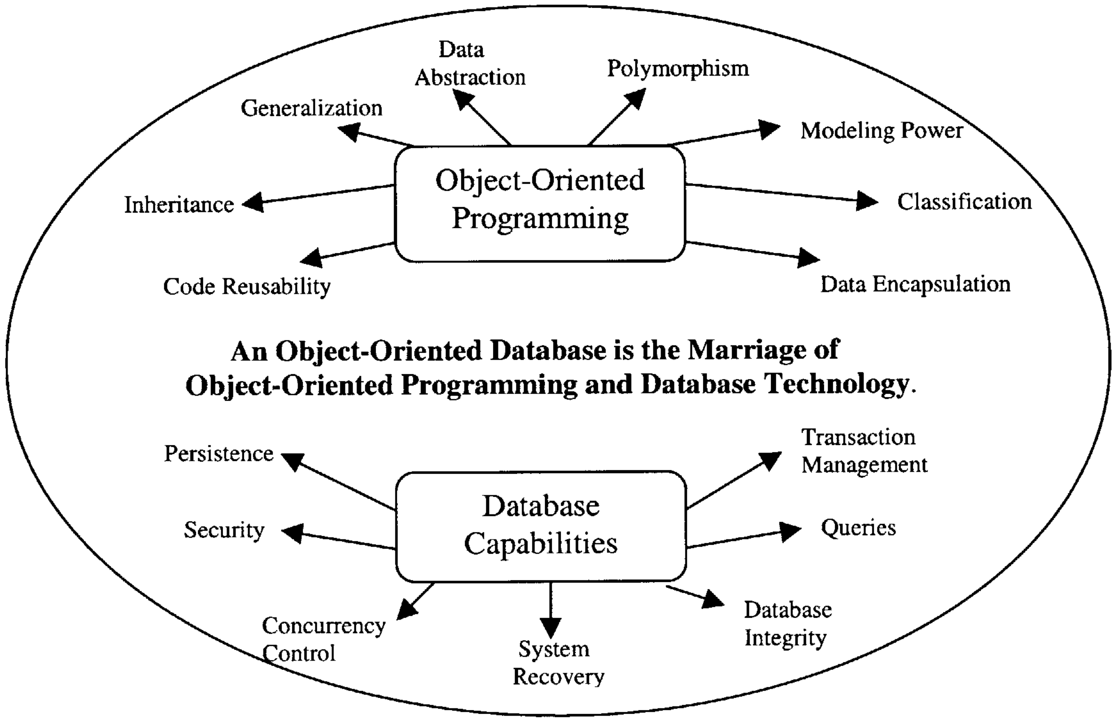


Figure 4-1: Makeup of an Object-Oriented Database [McFarland1999]

OODBMS provides all the database capabilities of a conventional database system including: persistence, transaction management, concurrency control, system recovery from crash, queries, security, and database integrity. As an object-oriented system, it also supports the following additional facilities [Ahmed1991, Nahouraii1991, Silberschatz1998]:

1. Data abstraction

The concept of packaging the data with the operations on the data is called data abstraction. The principle of data abstraction distinguishes the user of an object from its implementor. The implementation of the object is not visible to the user of an object. The users operate on the object by means of the operations which are made visible by the implementor. The concept of data abstraction provides a significant benefit in that the implementor can change the implementation of an encapsulated object without affecting the applications using it.

2. Classification

Similar entity instances are classified into types (or classes). The relationships between entity types and instances are known to the system and can be utilized to formulate queries which span data and schema.

3. Generalization

Similar entity types can be generalized into supertypes which capture their similarities. Existing types can be refined to create subtypes which inherit the properties and operations of their supertypes and may have their own specific properties and operations.

4. Code reusability

Code reusability is to use any predefined routines that already exist on the system. OODBMS provides a wide set of predefined types and their associate operations.

5. Data encapsulation

Encapsulation is the mechanism to implement the concept of data abstraction. It means to enclose in or as if in a capsule. An object is said to encapsulate both its data (state representation, attributes of the object are synonymous terms for data) and the operations on the data. Stated differently, data plus the operations on the data are enclosed in an object. Encapsulation describes the visibility of the data as well as the visibility of the operations on the data.

6. Inheritance

The concept of inheritance allows the creation of a new class from an existing class but perhaps with some changes (in terms of adding new operations and data, redefining existing operations etc.). So, inheritance provides a very powerful mechanism that allows the sharing of resources (data + operations) among classes.

7. *Polymorphism*

Polymorphism is defined as the ability of different objects belonging to different classes to respond differently to the same message. Consider the example, assuming there are two graphic objects: a square object and a triangle object. Sending a draw message to these objects would invoke different "draw" operations.

8. *Modeling power*

The objects and their interrelationships can be aligned very closely to the real-world objects and their interrelationships, making them ideal for simulation and design purposes.

After all, the flexibility and power of object-oriented programming combined with conventional database capabilities offer significant and wide spectrum of benefits in designing and developing complex systems. Powerful features such as inheritance, polymorphism, data encapsulation or code reusability enable us the creation, coordination, storage and manipulation of complex, highly interrelated objects efficiently.

Next, we will discuss the requirement of a DBMS for our collaborative work. Then, we will make a comparison of OODBMS and relational DBMS, and finally, explain why OODBMS possess exclusive advantages for our collaborative work.

4.2 Requirements of a DBMS for Collaborative Application Services

A primary characteristic of some applications like Computer Aided Design (CAD), Computer Aided Manufacturing (CAM) or Computer Aided Software Engineering (CASE) is the need to manage a group of designers working collaboratively on geographically distributed locations to complete a complex design by closely interacting among themselves and dynamically sharing data. For example, the manufacture of an aircraft requires the tracking of millions of interdependent parts that may be assembled in different configurations. Such environments necessitate powerful data modeling, sharing and transaction management tools, efficient communication protocols and a flexible framework for concurrency management of highly interactive transactions. The essential features of a database management system that are required for supporting such applications may be summarized as follows:

- Complex information modeling power
- Flexible transaction framework
- Runtime schema access/definition/modification
- Data abstraction and encapsulation
- Classification and generalization
- Data sharing capabilities
- Database compatibility and extensibility
- Concurrency control

4.3 Comparison of OODBMS and RDBMS

Object-oriented DBMS (OODBMS) and Relational DBMS (RDBMS) are the two most common DBMS in the market [Nahouraii1991]. We will discuss and compare each of these below.

Relational database design is a process of trying to figure out how to represent real-world objects within the confines of tables in such a way that good performance results and preserving data integrity is possible. Object database design is quite different. For the most part, object database design is a fundamental part of the overall application design process. The object classes used by the programming language are the classes used by the OODBMS. Because their models are consistent, there is no need to transform the program's object model to something unique for the database manager.

A significant difference between object-oriented databases and relational databases is that object-oriented databases represent relationships explicitly, supporting both navigational and associative access to information. As the complexity of interrelationships between information within the database increases, the greater the advantages of representing relationships explicitly. Another benefit of using explicit relationships is the improvement in data access performance over relational value-based relationships.

Relational systems are based on a simpler approach to data organization [Nahouraii1991], the *relation* or *table*, which allows a considerably clearer distinction between a logical and a physical data model. Relational systems offer a high degree of physical data independence and include powerful languages, even though they have limited expressive power. Physical data independence means that the physical storage of data is transparent to the user and may in principle be changed without also changing the logical view of the data. Relational languages are *set-oriented* (as opposed to record-oriented) and can thus be non-procedural or *declarative*. Set-oriented processing means that the tables of a relational database can be manipulated in their entirety by special operators; there is no need to iterate tuple by tuple through the relation.

The declarative nature and limited power (compared to a programming language) of the SQL language provides good protection of data from programming errors, and makes high-level optimizations, such as reducing I/O, relatively easy.

While database systems were mainly used in business and management applications initially, it was recognized that advantages could be gained from databases in scientific, technical, office and other fields as well. However, relational systems reach their limitations in applications like CAD, CASE or CAM which are collaborative in nature. OODBMS will be the best choice for these applications. We will enumerate the advantages of OODBMS in the next section.

4.4 Advantages of OODBMS for Collaborative Application Services

Based on our discussion in the previous section, there are distinctive advantages of using OODBMS to support our collaborative framework. We briefly enumerate these advantages as follows [Brown1991, Ishkawa1993, Lausen1998, Nahouraii1991]:

1. Complex Objects

OODBMS is unique in that the information being maintained is organized in terms of the real-world entities being modeled. This differs from relational database applications that require a translation from the real-world information structure to the table formats used to store data in a relational database. Normalizations upon the relational database tables result in further perturbation of the data from the user's perceptual viewpoint. OODBMS provides the concept of complex objects to enable modeling of real-world entities, which gives a much better feel for the mechanics and behavior of our collaborative environments.

2. Object Identity

OODBMS provides the concept of an object identifier (OID) as a means of uniquely identifying a particular object. OIDs are system generated and never change, even across application executions. Moreover, the OID is not based on the value stored within the object. This differs from relational databases, which use the concept of primary keys, which are based upon data stored in the identified row, to identify a particular row. The concept of OIDs makes it easier to control the storage of objects (e.g. not based on value) and to build links between objects (e.g. they are based on the never changing OID). It is vital to our collaborative applications involving complex and highly interrelated objects.

3. Runtime Schema Access/Definition/Modification

OODBMS makes use of a *database resident representation* of the schema for the database. The existence of such a representation, and a means of accessing it, provide our collaborative applications with direct access to schema information. Access to schema information will be useful in building custom tools that browse the structure and contents of a database, while modification and definition of the schema at runtime allows the development of dynamically extensible applications. Since collaborative environments are characterized by continual change, schema definitions are likely to be modified as designers arrive at better understanding of their design. OODBMS is flexible and provides extensive access and modification of schema.

4. Composite Objects

OODBMS provides the relationship mechanisms to connect composite objects. Composite objects are groupings of inter-related objects that can be viewed logically as a single object. Composite objects are used to model relationships that have the semantic meaning is-part-of. (e.g. rooms are part of a house). The mechanisms provide a powerful capability for our collaborative object design.

5. Distributed Client-Server Approach

As proposed in Chapter 1, ieCollab is composed of client-server distributed system architecture. OODBMS typically executes in a multiple process distributed environment. Server processes provide back-end database services such as management or secondary storage and transaction control. Client processes handle application specific activities such as access and update of individual objects. These processes may be located on the same workstation or on different workstations. Typically a single server will be interacting with multiple clients servicing concurrent requests for data managed by that server. A client may interact with multiple servers to access data distributed throughout the network. The distributed client-server approach will be essential for our collaborative applications performing lots of data updates.

6. Concurrency

Optimistic concurrency control mechanisms are based on the assumptions that access conflicts will rarely occur. Under this scenario, all accesses are allowed to proceed and, at transaction commit time, conflicts are resolved. OODBMS has incorporated the idea of optimistic concurrency control mechanisms for building applications that will have long transaction times. Handling of conflicts at commit time cannot simply abort a transaction, however, since one designer may be losing days or weeks of work. OODBMS provides techniques to allow multiple concurrent updates to the same data and support for merging these intermediate results at an appropriate time. Hence, it provides greater concurrency for our framework.

7. Transactions

Transactions are the mechanism used to implement concurrency. Within a transaction, data from anywhere in the database must be accessible. A feature found in many OODBMS products is to commit a transaction but to allow the objects to remain in the client cache under the expectation that they will soon be referenced again.

8. Impedance Mismatch

One of the main criticisms of relational database programming is the impedance mismatch between the data manipulation language (DML), normally SQL, and the application programming language. Relational database applications have an impedance mismatch, in that database access via the query language is table-based while application programming is individual value-based. Extra code and intellectual hurdles are required to translate between the two. A presumed benefit of OODBMS is that the application programming language and the DML are one and the same, hence eliminating the problem of impedance mismatch.

In short, there are significant and wide spectrums of advantages of choosing OODBMS for our collaborative transaction management framework.

4.5 Summary

In this chapter we have provided the characteristics of OODBMS, and their requirements for supporting collaborative application services. Based on our comparison of OODBMS and RDBMS, we notice the advantages of using OODBMS in the transaction management framework.

The next chapter presents our transaction management model for collaborative application services, and discusses some advanced transaction management functionalities required to support the framework.

Chapter 5

A Transaction Management Framework for Collaborative Application Services

5.1 Introduction

Today's software tools like CAD tools or software development provide only very limited support for collaborative transaction management framework [deBy1998]. A major conceptual problem in the transaction management framework is to ensure consistency criteria for the data concurrently processed by multiple users. Conventional database technology already provides mechanisms to absolutely guarantee consistency constraints by controlling the concurrent access of different users to shared data. Unfortunately, existing transaction management concepts are not suitable for supporting and controlling collaboration between users, because they are designed to fully isolate users from each other. In this chapter, we develop a transaction management model for collaborative applications in ieCollab environment that can support arbitrary collaboration schemes according to application needs.

The basic requirements of a transaction management framework for collaborative applications have been discussed in detail in Chapter 1. They are summarized here as follows:

- Flexible and efficient data sharing capabilities in the forms of grouped and shared transactions
- Nested transactions framework for task discretization
- Copying versus locking schemes
- Deadlock Prevention and Detection
- Effective protocols to handle conflicts or triggers among users
- Transaction checkpointing
- Semantic-based concurrency control to increase the level of concurrency

5.2 Basic Features in Transaction Management

The goal of transaction management is to maintain integrity of data while enabling execution of multiple concurrent transactions by various clients. A transaction should possess the following properties [Ahmed1991, Atluri2000, deBy1998, Elmagarmid1992, Gray1993, Reuter1993]:

- ***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 transaction should transform the system from one consistent state to another consistent state.
- ***Isolation:*** Each transaction should appear to execute independently of other transactions that may be executing concurrently in the same environment. The effect of executing a wet of transactions serially should be the same as that of running them concurrently. This requires two things:
  - ♦ During the course of a transaction, intermediate (possible 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. Database management systems usually implement this feature using locking.
- ***Durability:*** The effects of a completed transaction should always be persistent.

These properties guarantee that a transaction is never incomplete, the data in never inconsistent, concurrent transactions are independent, and the effects of a transaction is persistent. In order to implement all these properties, a transaction management system should be deployed which will include the following basic features:

- ***Transaction scheduling,*** which is responsible for initiating, queuing, executing, terminating or aborting transactions;
- ***Locking facilities,*** which controls the locking of items such as objects or classes to guarantee exclusive use of a data item to a current transaction;

- ***Deadlock management,*** which detects and resolves deadlock between transactions;
- ***Concurrency control,*** which ensures that concurrently executing transactions maintain database consistency;
- ***Recovery management,*** which provides facilities for database restoration from soft and hard system crashes;
- ***Security control,*** which protects data from accidental misuse.

We will first provide a brief review of how these transaction management features have been implemented in traditional database management systems and their limitations for collaborative application services. Subsequently, we will present enhanced functionalities required for supporting collaborative work.

5.3 A Review of Traditional Transaction Management Framework and their Limitations for Collaborative Application Services

For our study of collaborative transaction management framework, we will briefly discuss transaction management functionalities in traditional database management systems and their limitations for supporting collaborative work.

Traditional transaction management was mainly developed for business data processing/administrative applications [Reuter1993]. In such environments transactions are assumed to be rather short (typically at most a few seconds). Transaction management provides exactly one type of transactions which is flat and works directly on the data of the database. Moreover, it ensures strict isolation with the consequence that transactions are not allowed to exchange data.

The principal transaction management features provided for such systems are briefly discussed below.

1. ***Nested and grouped transactions***. Traditional database systems do not support the idea of nested transactions, where a transaction is hierarchically sub-divided into sub-transactions. They also do not support for grouped transactions, where multiple users share data and perform operations as a single transaction unit.

2. ***Serializability***. Given an interleaved execution, serializability requires us to find a serial execution in which transactions read the same values. Serializability is based on the assumption that individual concurrent transactions run ignorant to each other and do not interact in the middle of their execution. This is however unsuitable for our collaborative application transactions, because collaborative application is based on the fact that unit of work must interact, so that the results are usable together.

3. ***Locking protocols***. Traditional databases normally only provide three types of locks: read, write and exclusive. Locking is generally not communicative, i.e. the system does not inform the user of the lock status of the object (e.g. whether it is free, or locked by someone, in which mode, etc.). Thus, a user may wait indefinitely for a

lock on an object without being informed in advance by the system of the lock status of the object. Moreover, a user holding a lock on object is not informed if anyone else is waiting for that object. Furthermore, traditional databases do not support the notions of shared locks between collaborating users or allow two or more users to update the same object by acquiring write locks on the object simultaneously.

4. ***Conflict handling***. In traditional transaction management frameworks, conflict handling is generally poor. Generally, in the event of deadlocks, one or more transactions are arbitrarily aborted by the system without prior notification to the clients involved. Since long collaborative transactions may represent a great deal of work, an abort without notification may induce heavy cost.

Although the above functionalities are sufficient for financial and business applications, since collaborative transactions are often of long duration, it is highly inappropriate for such work because it inhibits data sharing, results in reduced concurrency and intolerably long waits.

It is therefore necessary to make careful considerations for the design of a more flexible and efficient transaction management system that allows users on collaboration to arrive at a complex design without requiring to wait over a long duration.

5.4 A Transaction Management Model for Collaborative Application Services

As discussed in the previous section, it has been found that the traditional transaction management concept has limited applicability in collaborative work. For collaborative applications such as CAD, CAM, office automation, publication environments or software development environments, transactions are usually very complex, need to access many data items and reside in the system for long duration. For example, a software engineering project is usually performed jointly by different groups of people; each person is responsible for a part of the project within his/her group. People of the same group have to cooperate in order to achieve a good software development. One means of cooperating is by exchanging information through shared data items. To do this, people (or their corresponding transactions) need to access the shared data items alternatively. People in the same group have to work collaboratively on a specific task. This results in the notion of cooperating grouped transactions. The functional division of a software engineering project is shown as a linear sequential model in Figure 5-1 [Pressman1997].

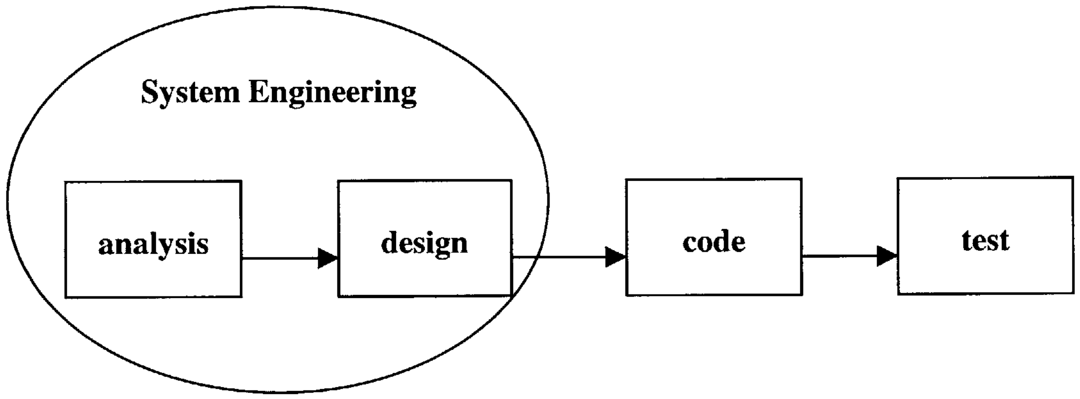


Figure 5-1: Linear Sequential Model for Software Engineering [Pressman1997]

Software engineering is simplified into four sequential approaches respectively: software requirement analysis, design, code generation and testing. Requirement analysis group documents the requirements for both the system and the software. The design group then translates requirements into a representation of the software that can be assessed for quality before code generation begins. The code generation group performs a translation form the design into a machine readable form. Finally, once code has been generated, program testing can begin.

In software engineering environments, support is needed for long-lived transactions and cooperative transactions. It is also desirable to be able to support various levels of cooperation dependent on the particular environment.

We are going to present our model using nested active transactions with user-definable correctness conditions. Users can define appropriate correctness conditions [Elmagarmid1992] based on the following:

- ***Conflicts:*** Operations that are not allowed to execute concurrently.
- ***Patterns:*** Sequences of functions that must occur. For example, a pattern of edit-compile-link-test may be required when editing a source code file.
- ***Triggers:*** Actions that are taken when a request begins or ends.
- ***Commit or Abort Semantics:*** Actions to be taken when a transaction ends.

Since user-definable correctness conditions are used, *semantic-based concurrency control* can be user to increase the level of concurrency. It is also possible to allow a higher degree of cooperation between different transactions, and to allow various levels of cooperation. This also helps to support long-lived transactions.

To summarize our model for a collaborative environment, we will emphasize the following issues:

- Grouped transactions are formed to facilitate data sharing among members within a specific group.
- Members in a group work collaboratively by sharing data item and participate in shared transactions.
- Nested active transactions are formed to represent specific groups of a project.
- Enabling users to define correctness conditions such as conflicts, patterns, triggers or commit/abort semantics allow for greater flexibility.
- Semantic-based concurrency control is used to increase the level of concurrency and allow high level of cooperation between transactions.

5.5 Transaction Management Functionalities for Collaborative Application Services

In the previous section, we have briefly discussed our transaction management model for collaborative application services. In this section, we will describe our proposed model for transaction management [Atluri2000, Cellary1988, deBy1998, Reuter1993].

5.5.1 Grouped Transactions

In our model, users (and application programs) operate in the context of a group structure as shown in Figure 5-2 – a hierarchy of groups. The top-most node of the hierarchy represents the software product organization. The lowest node represents individual groups on a specific task of a specific product in the product life cycle.

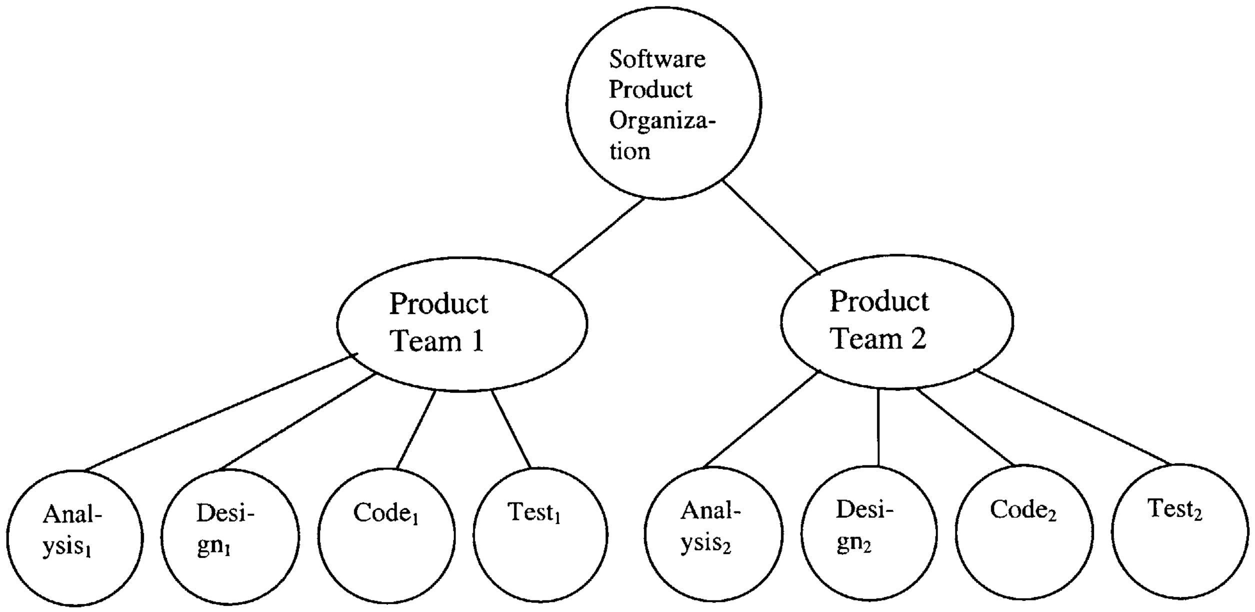


Figure 5-2: A Hierarchy of Groups

A group's protocol regulates the sharing of objects and concurrent operations by the immediate members of that group, whether they are individual users or sub-groups. For example, imagine a software product organization that is divided into two different product teams, T1 and T2. The top level protocol might indicate that all operations on a system s from T1 are to be rejected while s is being tested by T2, and vice versa. At the next lower level, the protocol for T1 might allow two different members of T1 to test the system concurrently, but not to modify the system while it is being tested. The protocol for T2 might enforce different policies for interactions among its members.

Each group in the hierarchy has a (local) transaction manager, which has two main parts: 1) a group-specific protocol, and 2) a uniform capability (over all groups) for coordinating with neighboring transaction managers. This division of responsibilities keeps the protocols independent of each other and allows each protocol-writer to focus on interactions within one group.

5.5.2 Specifying Protocols

In our model, transaction management is based on each organization's specification of 1) how concurrent functions can interleave; and 2) required procedural patterns. In other words, the model supports a scheme in which the transaction manager does not have a single, built-in notion of correctness, but rather is programmed to accept its own application-specific protocols. Our goal is to accommodate the variety of protocols that would be useful for our collaborative framework.

The protocols would be used to describe:

- ***Conflicts:*** Requests that are not allowed to execute concurrently because the functions, the arguments, or both conflict. For example, no member of group G1 may test a code module m while a member of G2 is editing m. Once the member of G2 is finished editing, the member of G1 may be allowed to proceed.
- ***Patterns:*** Sequences of requests that must occur before a transaction may commit, or before a request can be accepted. For example, immediately after linking a system, a test suite must be executed to determine if that system still works. If the system were linked, and the test suite were not executed, then other requests, including the request to end the transaction (commit) would be rejected.
- ***Triggers:*** Additional requests that are submitted when a request begins or ends. For example, before an update is performed to a system module, a copy is made to a separate directory to isolate the released system from the experimental version; or after a modification has been made and tested, the new code is merged back into the release directory.

5.5.3 Deadlock Prevention/Detection

Deadlock is caused by a cycle of "wait-for" among requests. In the simplest case, deadlock occurs when a request r1 has exclusive access to (a lock on) an object o1 and is blocked because a sub-request awaits exclusive use of another object o2; at the same time another request r2 has exclusive access to o2 and is blocked because a sub-request awaits exclusive user of o1. Neither transaction can release its lock so that the other transaction can be dequeued, and the two remain deadlocked.

A potential deadlock is detectable at the lowest common transaction manager for the transactions involved. Queuing occurs at the same node at which locks are granted (i.e. where usage would be tested in the history. This can only be at the node that controls the requests of both transactions involved in deadlock. It is at this node that queuing is performed and analyzed and potential deadlock is detected and dealt with.

The potential deadlock, then, is visible to a single node, which can autonomously determine what response to take. The protocol of a transaction manager can prevent deadlock by rejecting the request that would cause it.

5.5.4 Copying versus Locking

The transaction manager makes the objects accessible and yet restricts requests from other member transactions that may interfere. Protocols accomplish this by ensuring that some controlled copy of the object is accessed by the procedures referenced in the transactions.

As an example, an object may have just one copy, shared by all users and groups in the system (and therefore, controlled by the root node). This approach is often associated with protocols that use histories to obtain the effect of locking. Alternatively, a protocol may produce copies of objects for its children to share, thus providing "less public" copies whose modifications are not seen by members of the larger group until made available by the user's "checkpointing" those objects. Copies may exist at only some lower levels in the hierarchy. Our model makes it easy to define protocols where different groups use different nodes.

The "locking" approach avoids dangerous concurrent operations on the same copy of an object. It queues or rejects requests that conflict with some request that is currently being performed; the blocked request may be scheduled at a later time when it is no longer dangerous. This approach eliminates the need to make multiple copies at the cost of making transactions wait. There are cases in which the waiting approach is desirable, particularly where there is no way to merge changes from multiple copies or where very large objects are involved and therefore copying and merging is too costly.

The "copy" approach avoids dangerous concurrent operations on the same object by replicating the state of the object (its representation) for each group member who accesses the object. This approach is optimistic since it allows requests that are logically conflicting to proceed simultaneously on their respective copies. It requires reconciliation of changes to the locally cached copies with the next higher copy (e.g. replace, merge, etc.) when each group member "checkpoints" the object (or commits) or otherwise indicates that his or her modifications are complete.

5.5.5 Checkpointing

To checkpoint an object or a set of objects is to make the effects of a transaction's operations on those objects permanent before they are committed. Transaction managers can provide "checkpoint" functions that guarantee that in the case of the subsequent failure of the transaction, those effects can be recovered.

The protocol for a group determines how the group reacts when one of the objects shared by its members is checkpointed. For example, if the protocol has triggered creation of a private copy for the requester, it might trigger an update to its master copy with the updated values of the group's private (locally cached copy), in order to share that value with other groups. Likewise, checkpointing may trigger messages to notify others that the requester has completed modifications or tests.

5.5.6 Shared Transactions

Shared transactions represent a mechanism for enabling the greatest flexibility between users. In our case, two or more users are involved in a single, atomic, multi-user transaction. This enables unrestricted data sharing among multiple users instead of having to cross transaction boundaries. For example, users involving in a shared transaction for a particular task may all acquire shared write locks on the object concerned. Each user may update the object locally, and each update sends notifications to other users who respond to them properly. When the users agree on a common task, a stable version of the object is passed back to the parent database for sharing with other users.

5.5.7 Nested Transactions

Nested transactions extend traditional transactions by applying concurrency control and recovery concepts within transactions as well as among them. This permits safe concurrency within transactions and allows transactions to fail partially, in a controlled manner. Any transaction can have child transactions nested within it. Nested transactions can, in turn, have their own child transactions. Therefore, there can be a whole hierarchy of transactions associated with a given transaction.

A nested transaction is a set of partially ordered atomic read and write operations as well as atomic transactions. This permits modular and concurrent composition of transaction programs. For example, considering a bank account, a transfer transaction can be composed by using the existing deposit and withdraw transaction programs. Furthermore, since the deposit and withdraw transactions will operate on different objects, they can be executed concurrently. Even if the internal transactions conflict with each other, the execution atomicity of these transactions will be ensured y the synchronization algorithms used to implement nested transaction execution. Nested transactions, therefore, provide a modular approach to realize both intra-transaction and inter-transaction parallelism.

5.5.8 Semantics-based Concurrency Control

One of the key requirements of a collaborative application environment is a flexible concurrency control mechanism that allows a high degree of parallelism and data sharing between concurrently executing transactions and maintains database consistency. Semantics-based concurrency control provides a way to ensure database consistency as well as great concurrency.

The idea behind semantics-based concurrency control is as follows. By abstracting from the low-level details (i.e. the concrete implementation in the database), and by exploiting the high-level semantics of data objects and operations, certain conflicts that might occur between decompositions of the operations into elementary read/write sequences can be ignored. The most commonly used approach to capture the semantics of data objects is to specify commutativity relations among operations defined on single data objects. Informally, two operations commute (do not conflict) if their effects on the state of the object and their return values are the same regardless of their execution order. This ensures that no transaction can observer a difference between both execution orders.

When a transaction requests the execution of an operation, this request can be granted if the operation commutes with all other operations of uncommitted transactions. This policy ensures the semantic serializability of concurrent transactions – not at the level of disk page accesses but on the level of higher-level operations.

Semantics-based concurrency control does not support collaboration directly but it achieves a high degree of concurrency. Thereby, it reduces the probability of long waits or aborts which is important to our collaborative environment where transactions are typically of long duration.

5.6 Summary

In this chapter, we have presented a model for collaborative application environment and discussed the transaction management framework for facilitating coordination and concurrency using OODBMS for such work. Key issues that have been emphasized include shared and grouped transactions among users, deadlock prevention, transaction checkpointing, comparison of copying versus locking, nested transactions for better management, greater flexibility in concurrency by semantics-based concurrency control, and specifying protocols to support our model. The limitations of traditional DBMS for our collaborative environments have also been overviewed.

The next chapter will discuss the implementation of an OODBMS for our database layer using ObjectStore PSE Pro.

Chapter 6

Implementation Issues

6.1 Introduction

One of the most challenging tasks in object-oriented database model is to implement a robust persistence architecture, since object-oriented database model usually consists of a large number of distinct object types, with complex internal structures and navigational relationships between objects. In other words, good object-oriented designs are not "flat", and they cannot be modeled easily using only simple, discrete data structures.

This chapter describes the implementation of persistent Java objects to our database layer in ieCollab using an OODBMS, namely, *ObjectStore PSE Pro* [Exceloncorp2000] on a platform Microsoft Windows NT. Workstation 4.0. Since ObjectStore PSE Pro directly supports the Java programming language, it can essentially be hosted in any platform.

ObjectStore PSE Pro is particularly chosen as the OODBMS for our implementation because of the following properties:

1. It is compact, portable, and administration-free.
2. It supports CORBA architecture and hence, is *ideal* for our three-tier, thin client system architecture.
3. Some essential features provided include:
    - *100% Pure Java* – runs everywhere
    - *Pure object DBMS* – stores Java data in native format, as objects
    - *Unlimited extensibility* – stores any Java object type
    - *Seamless Java interface* – manages data using native Java
    - *Data navigation* – performs orders of magnitude faster than RDBMS
    - *Atomic transaction commit and rollback* – ensures data integrity
    - *Transaction recovery* – restores data from system crashes
    - *Rich query and JDK 1.2 collection support* – enables data access with native Java
    - *Persistent garbage collection* – keeps database compact
    - *Flexible transaction model* – provides multiple threads

After giving a brief introduction of ObjectStore PSE Pro, in the following section, we will enumerate some of the major advantages of implementing ObjectStore PSE Pro for our collaborative transaction management framework.

6.2 Advantages of ObjectStore PSE Pro for Collaborative Application Services

As mentioned in the previous section, ObjectStore PSE Pro is a pure Java, pure object database for embedded database applications. There are several distinctive advantages of implementing ObjectStore PSE Pro for our collaborative framework. These are enumerated as below:

1. Superior Scalability

ObjectStore provides a distributed multi-tiered architecture that leverages replication and caching of both data and services to meet unprecedented scalability demands. This is vital to our distributed thin client system architecture in ieCollab. Distributed caches provide local, in-memory data access to the various services and components that make up the application, and both data and processing are distributed across each application tier. This cuts down on repetitive and unnecessary network traffic, removes database server bottlenecks, and allows scaling without adding expensive hardware.

2. Ease of Use

ObjectStore PSE Pro provides an easy to use interface for storing and retrieving Java objects. We can define persistence-capable Java classes, and their fields and methods, in the same way as transient Java classes. We use standard Java constructs to create and manipulate both persistent and transient instances.

In addition, ObjectStore PSE Pro for Java API provides database features that allow us to

- Create, open and close databases
- Start and end transactions
- Store and retrieve persistent objects

3. *Shorter time to market*

ObjectStore enables rapid deployment by providing seamless support for object-oriented development languages and ready-to-use object managers for managing rich multimedia content such as audio, video, images, clipboard, text, and others which would be inherent in our distributed applications in ieCollab. ObjectStore object managers provide a full set of off-the-shelf, reusable extended data types for handling all those multimedia content. We can easily extend these data types or define new data types, and they can encapsulate their own methods to define the processing that goes with the data.

4. *Increase productivity, decrease development cost*

Storing objects in a flat file or RDBMS usually involves writing complex code to map the application's object model to the sequential or relational model. In many cases, this coding logic composes over half of the application's source code. ObjectStore eliminates the need for complex mapping logic by directly binding to Java in order to make objects persistent and store them in their natural representation – as *objects*, instead of rows and columns. By drastically reducing the amount of code required to manage persistent objects, ObjectStore improves application quality and performance, and reduces development and maintenance costs. This facilitates our comprehensive development of collaborative distributed applications for different market needs.

5. Standards

ObjectStore supports CORBA, JavaBeans, JDO, SQL standards (and others), hence it will integrate seamlessly with our ieCollab computing infrastructure.

6. Built-in Support and Recovery Features

ObjectStore provides a full set of reliability and high availability options for delivering fully continuous operation that ensures the integrity and reliability of data in our applications.

- ObjectStore's nonstop features include automatic recovery, on-line backup, roll-forward capability, failover, and replication.
- ObjectStore's failover support allows our client applications to automatically reconnect to standby servers in the event of a primary server failure with no loss of service.
- Replication can be used to provide shared access to data from our geographically distributed applications, resulting in improved performance and resiliency from site failures.

The following sections describe the application development issues of ObjectStore in our collaborative transaction management framework. We will focus on issues regarding transaction and concurrency control, transactions and recovery, persistence, runtime schema access/definition/modification, and composite objects.

6.3 Transaction and Concurrency Model

ObjectStore PSE Pro provides a transaction implementation that supports a traditional *readers/single writer concurrency control* policy for the access of individual objects. PSE Pro supports four types of transactions: *update, read-only, update non-blocking* and *read-only non-blocking.*

If the *update* mode is selected, the application is permitted to modify data within the current transaction's open database. A write lock is requested, and if not used by another session under the same Java VM, granted. The transaction request blocks until a write lock is granted. The *read-only* transaction allows the application to read but not modify data in the open database for the current transaction. If there are no sessions within the same Java VM owning or waiting for an update lock, a shared read-lock is granted; otherwise, the application blocks until a read-lock is available. An *update non-blocking* transaction is similar to the update transaction, without the blocking. If a write lock is not immediately available the transaction operation exceptions out. The *read-only non-blocking* transaction is similar to the read-only transaction, without the blocking. If a read lock is not immediately available the transaction operation exceptions out.

PSE Pro can support multiple sessions within the same Java VM process. Each session can access the same database through a read-only transaction, and simultaneously engage separate update transactions against different databases. Only one session at a time can open an update transaction against a particular database.

6.4 Transactions and Recovery

PSE Pro can recover an application failure or system crash. If a failure prevents some of the changes in a transaction from being saved to disk, PSE Pro ensures that none of that transaction's changes are saved in the database. When we restart the application, the database is consistent with the way it was before the transaction started.

6.5 Persistence

PSE Pro provides persistence on an object basis, through an enhanced form of Java object serialization. The Java object type and safety properties are maintained in the serialized form and serialization only requires per class implementation for special customization. Objects are created as transient objects, and are promoted to persistent objects if they are added to an existing persistent object.

6.6 Runtime Schema Access/Definition/Modification

ObjectStore database contents and object types can be retrieved during runtime either through an inspection utility, or through API method calls. Information available includes:

- Name of the database
- Name and number of each type of object in the database
- Total size in bytes occupied on the disk by each type of object
- Number of destroyed objects

6.7 Composite Objects

ObjectStore supports class attributes whose type are another class, thus supporting the building of composite objects. Objects referred to by a persistent object are themselves persistent, unless the reference was keyworded as transient. Delete and lock operations apply to these persisted through reference objects.

6.8 Implementation Details of Persistent Java Objects

This section discusses the core concepts involved with writing a PSE Pro application. It provides implementation details of persistent Java objects using ObjectStore PSE Pro which serves as an example of Java data management in object-oriented database environment for collaborative work.

Before we can create and manipulate persistent objects with PSE Pro, we must perform the following operations:

- Create a session
- Create or open a database
- Start a transaction

6.8.1 Creating Sessions

To use ObjectStore PSE Pro, our application must create a session. A session is the context in which PSE Pro databases are created or opened, and transactions can be executed. Only one transaction at a time can exist in a session. PSE Pro allows us to create multiple sessions and thus have multiple concurrent transactions in a single Java VM process.

Any number of Java threads can participate in the same session. Each thread must join a session to be able to access and manipulate persistent objects. To create a session we call the **Session** constructor and specify the host and properties. The method signature is

```
public static Session create(String host,
    java.util.Properties properties)
```

A thread can join a session with a call to **Session.join().** For example:

```
/* Create a session and join this thread to the new session. */
session = Session.create(null, null);
session.join();
```

PSE Pro ignores the first parameter in the create() method. We can specify null. The second parameter specifies null or a property list.

6.8.2 Creating, Opening, and Closing Databases

Before we begin creating persistent objects, we must create a database to hold the objects. In subsequent processes, we open the database to allow the process to read or modify the objects. To create a database, we call the static **create()** method on the **Database** class and specify the database name and an access mode. The method signature is

```
public static Database create(String name, int fileMode)
```

The **initialize** method in the **UserManager** class shows an example.

```
public static void initialize(String dbName)
{
/* Other code, including creating a session and joining thread to session*/
    /* Open the database or create a new one if necessary. */
    try {
        db = Database.open(dbName, ObjectStore.UPDATE);
    } catch (DatabaseNotFoundException e) {
        db = Database.create(dbName, ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);
    }
```

The **initialize()** operation first creates a session and then joins the current thread to that session. Next **initialize()** tries to open the database. If the database does not exist, DatabaseNotFoundException is thrown and is caught by **initialize(),** which then creates the database. **initialize()** also stores a reference to the database instance in the static variable **db**.

The **Database.create()** and the **Database.open()** methods are called with two parameters. In both methods, the first parameter specifies the pathname of a file. In the **create()** method, the second parameter is a UNIX-style protection number. In the **open()** method, the second parameter specifies the access mode for the database, that is, **ObjectStore.UPDATE** or **ObjectStore.READONLY**.

Shutting down

The **UserManager.shutdown()** method shows an example of how to close a database and how to terminate a session.

```
/**
* Close the database and terminate the session.
*/
    public static void shutdown() {
        db.close();
        session.terminate();
    }
```

6.8.3 Starting Transactions

We create, destroy, open, and close a database outside a transaction. We access and manipulate objects in a database inside a transaction. Therefore, a program must start a transaction before it can manipulate persistent data. While the transaction is in progress, a program can read and update objects stored in the open database. The program can choose to commit or abort the transaction at any time.

Committing transactions

When a program commits a transaction, PSE Pro updates the database to contain the changes made to persistent data during the transaction. These changes are permanent and visible only after the transaction commits. If a transaction aborts, PSE Pro undoes (rolls back) any changes to persistent data made during that transaction.

Purpose of transactions

In summary, transactions do two things:

- They mark off code sections whose effects can be undone.
- They mark off functional program areas that are isolated from the changes performed by other sessions or processes (clients). From the point of view of other sessions or processes, these functional sections execute either all at once or not at all. That is, other sessions or processes do not see the intermediate results.

Creating transactions

To create a transaction, insert calls to mark the beginning and end of the transaction. To start a transaction, call the **begin()** method on the **Transaction** class. This returns an instance of **Transaction** and we can assign it to a variable. The method signature is

```
public static Transaction begin(int type)
```

The type of the transaction can be **ObjectStore.READONLY** or **ObjectStore.UPDATE**.

Ending transactions

PSE Pro provides the **Transaction.commit()** method for successfully ending a transaction. When transactions terminate successfully, they commit, and their changes to persistent objects are saved in the database. The **Transaction.abort()** method is used to unsuccessfully end a transaction. When transactions terminate unsuccessfully, they abort, and their changes to persistent objects are discarded.

When an application commits a transaction, PSE Pro saves and commits any changes in the database. It also checks to see if there are any transient objects that are referred to by persistent objects. If there are, and if the referred-to objects are persistence-capable objects, PSE Pro stores the referred-to objects in the database. This is the process of transitive persistence, also called persistence by reachability.

The default commit operation makes all persistent objects inaccessible outside of the transaction's context. After we commit a transaction, if we want to access data in the database, we must start another transaction and navigate to the object again from a database entry point. There are optional commit modes that allow us to retain the objects so that you can access them outside of a transaction or in a different transaction.

6.8.4 Storing Objects in a Database

Objects become persistent when they are referenced by other persistent objects. The application defines persistent roots and when it commits a transaction, PSE Pro finds all objects reachable from persistent roots and stores them in the database. This is called persistence by reachability and it helps to preserve the automatic storage management semantics of Java.

Example of Storing Objects in a Database

For example, consider the **subscribeNewUser()** method below, which adds a new user to the database.

```
public static int subscribe(String name, String email)
        throws PersonalizationException
    {
        Transaction tr = Transaction.begin(ObjectStore.UPDATE);
        /* First check to see if the user's name is already there. */
        if (allUsers.get(name) != null) {
            tr.abort(ObjectStore.RETAIN_HOLLOW);
            throw new PersonalizationException("User already there: " + name);
        }
        /* The user name is not there so add the new user;
             first generate a PIN in the range 0..10000. */
        int pin = pinGenerator.nextInt() % 10000;
        if (pin < 0) pin = pin * -1;
        User newUser = new User(name, email, pin);
        allUsers.put(name, newUser);
        tr.commit(ObjectStore.RETAIN_HOLLOW);
        return pin;
    }
```

The application checks whether the user name is already defined in the database. If the name is defined, PSE Pro throws PersonalizationException. If the name is not already defined, the application creates a new user, adds that user to the **allUsers** collection, and commits the transaction. Since the **allUsers** collection is already stored in the database, PSE Pro stores the new user object in the database when it commits the transaction.

6.8.5 Accessing Objects in a Database

After an application stores objects in a database, the application can use references to these objects in the same way that it uses references to transient objects. An application obtains initial access to objects in a database through navigation from a root or through an associative query. An application can retain references to persistent objects between transactions to avoid having to obtain a root at the start of each transaction.

Example of Using a Database Root

To access objects in a database, we must start a session, open the database and start a transaction. Then we can obtain a database root to start navigating the database. For example, in our earlier example of storing objects in a database, we obtain the "**allUsers**" root to obtain **User** objects.

```
allUsers = (Map) db.getRoot("allUsers");
```

Example of Using References

Consider again the **subscribe()** method in our earlier example. The first part of this method protects against storing a duplicate name by checking whether the user's name is already in the database. For example:

```
/* First check to see if the user's name is already there. */
    if (allUsers.get(name) != null) {
        tr.abort(ObjectStore.RETAIN_HOLLOW);
        throw new PersonalizationException("User already there: " + name);
    }
```

Since the class variable **allUsers** references the **allUsers** collection, the application can use the standard Java **Map.get()** method to check if the name is already stored. The same code would work for a persistent or transient collection.

6.8.6 Deleting Objects

When we delete objects in PSE Pro, we must

- Disconnect objects from their relationships and associations
- Destroy the object so that it is removed from the database

Example of Deleting an Object

For example, to remove users from the database, the application calls the **unSubscribe()** method.

```
public static void unsubscribe(String name)
    throws PersonalizationException
{
        Transaction tr = Transaction.begin(ObjectStore.UPDATE);
        User user = (User) allUsers.get(name);
        if (user == null) {
            tr.abort(ObjectStore.RETAIN_HOLLOW);
            throw new PersonalizationException ("Could not find user: " + name);
        }
        /* remove the user from the allUsers collection, and
         * remove all of the users interests from the allInterests collection */
        allUsers.remove(name);
         /* finally destroy the user and all its sub-objects */
        ObjectStore.destroy(user);
        tr.commit(ObjectStore.RETAIN_HOLLOW);
    }
```

First, our application ensures that the user exists in the **allUsers** collection. If the user does not exist, PSE Pro throws an exception. Next, the application calls the **remove()** method to remove the user from the **allUsers** collection. This disconnects the user from the set of users, which means that this user is no longer reachable and can now be removed from the database. Finally, to remove the user from the database, the application calls **destroy()** on the **User** object.

The above examples demonstrate the implementation concepts of persistent Java objects involved with writing a PSE Pro application which provides an overview of Java data management of ObjectStore PSE Pro in object-oriented database for collaborative work.

6.9 Summary

In this chapter we have presented the implementation issues of ObjectStore PSE Pro for supporting collaborative application services. The essential features and advantages of ObjectStore PSE Pro create extensive values to our transaction management framework. Core concepts involved with writing a PSE Pro application are discussed with concrete examples provided to demonstrate the implementation issues of object-oriented database environment using ObjectStore PSE Pro.

The next chapter summarizes our study for transaction management on collaborative application services. It also outlines some open issues and directions for future research and development.

Chapter 7

Conclusions

7.1 Summary

This study has provided a framework for supporting collaborative application services using an object-oriented database management system, called ObjectStore PSE Pro. It forms an integral component of an ongoing collaboration project at MIT, called ieCollab, which aims at addressing coordination and communication problems among geographically distributed teams in collaborative application services. A transaction management model has been proposed with an emphasis on the following issues:

- Grouped transactions
- Shared transactions
- Semantics-based concurrency control
- Copying versus locking
- Nested transactions
- Specifying protocols

- Transaction checkpointing
- Deadlock Prevention and Detection

The limitations of traditional DBMS for supporting our collaborative work have also been discussed.

Some of the significant advantages of implementing ObjectStore PSE Pro for our transaction management framework and the application development issues of ObjectStore have been addressed. Finally, the implementation details of persistent Java objects using ObjectStore PSE Pro for Java data management are illustrated.

7.2 Future Transaction Management Research and Development

In our ieCollab project, a particular transactional framework was implemented where the transaction model was fixed. Nevertheless, this kind of approach is not feasible if many models are needed, since the system may not support different kinds of transaction models.

A more complicated transactional framework, a kind of meta-level transactional framework should support indeed requirements of different transaction models. It should also support the specification of individual transactions obeying a particular model and their execution. A requirement for such an environment, which goes clearly beyond a simple framework, is the availability of tools to check the compatibility of different transaction models. This is important in cases where the same computation can use different models in different parts, or the same data can be accessed by transactions obeying different transaction models. These issues are for further research.

In short, future research on transaction management frameworks is challenging and requires high expertise and close cooperation of transaction management specialists, software engineers, experts of formal specification approaches, and application specialists. Although it is a difficult task, we expect to see the functioning and integrated transaction management frameworks in the near future.

Bibliography

[Adya1994]	Adya, A., "Transaction Management for Mobile Objects using Optimistic Concurrency Control", M.S. Thesis, January, 1994, Massachusetts Institute of Technology
[Ahmed1991]	Ahmed, S., "Transaction and Version Management in Object-Oriented Database Management Systems for Collaborative Engineering Applications", M.S. Thesis, January, 1991, Massachusetts Institute of Technology
[ASP2000]	ASP Industry Consortium, *http://www.aspindustry.com/,* 2000
[Atluri2000]	Atluri, V., Jajodia, S. and George, B., "Multilevel Secure Transaction Processing", Kluwer Academic Publishers, 2000
[Bell1992]	Bell, D. and Grimson, J., "Distributed Database Systems", International Computer Science Series, Addison-Wesley Publishing Company, 1992
[Brown1991]	Brown, A., "Object-Oriented Databases: Applications in Software Engineering", McGraw-Hill Book Company, 1991
[Castano1995]	Castano, S., Fugini, M., Martella, G. and Samarati, P., "Database Security", Addison-Wesley Publishing Company, 1995
[Cellary1988]	Cellary, W., Gelenbe, E. and Morzy, T., "Concurrency Control in Distributed Database Systems", Studies in Computer Science and Artificial Intelligence, Elsevier Science Publishers B.V., 1988
[Chen2000]	Chen, H., "ieCollab transaction management design specification v1.0", March, 2000
[Chung1998]	Chung, P. E., Bell Laboratories, "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer", January 1998
[Dan1992]	Dan, A., "Performance Analysis of Data Sharing Environments", An ACM Distinguished Dissertation 1991, The MIT Press, 1992
[deBy1998]	de By, R. A., Klas, W. and Veijalainen. J., "Transaction Management Support for Cooperative Applications", Kluwer Academic Publishers, 1998

[Elmagarmid1992] Elmagarmid, A.K., "Database Transaction Models for Advanced Applications", Morgan Kaufmann Publishers, Inc., 1992

[Freitas1998] Freitas, A. A. and Lavington, S. H., "Mining Very Large Databases with Parallel Processing", Kluwer Academic Publishers, 1998

[Gray1993] Gray, J., "The Benchmark Handbook for Database and Transaction Processing Systems", Morgan Kaufmann Publishers, Inc., 1993

[Ishkawa1993] Ishkawa, H., "Object-Oriented Database System: Design and Implementation for Advanced Applications", Springer-Verlag, 1993

[Konduri1999] Konduri, G., "A Collaborative Environment for Distributed Web-based CAD", M.S. Thesis, February, 1999, Massachusetts Institute of Technology

[Lausen1998] Lausen, G. and Vossen, G., "Models and Languages of Object-Oriented Databases", Addison-Wesley Longman Ltd., 1998

[McFarland1999] McFarland, G., Rudmik, A. and Lange, D., DACS Technical Reports, http://www.dacs.dtic.mil/techs/, January 1999

[Nahouraii1991] Nahouraii, E. and Petry, F., "Object-Oriented Databases", IEEE Computer Society Press, 1991

[OD2000] Object Design Inc., "Embedding ObjectStore PSE Pro for Java, The Pure Java, Pure Object Database", An Object Design Technical Brief, 2000

[Pressman1997] Pressman, R. S., "Software Engineering, A Practitioner's Approach", The McGraw-Hill Companies, Inc., 1997

[Reuter1993] Reuter, A. and Gray, J., "Transaction Processing: Concepts and Techniques", Morgan Kaufmann Publishers, 1993

[Silberschatz1998] Silberschatz, A., Korth, H. F. and Sudarshan, S., "Database System Concepts", 3rd Edition, WCB McGraw Hill, 1998

[Exceloncorp2000] Produdct Overview, ObjectStore PSE Pro Enterprise Edition, http://www.exceloncorp.com/, 2000

[Sun2000] "JDBC Data Access API", http://java.sun.com/products/jdbc/, 2000

List of Appendices

Appendix A

Requirements Specification
For ieCollab Version 2
Transaction Management

Specification Version 1.6

Requirement Analysis Team

Update from Version 1.5 by Bharath Krishnan, Alan Ng
Date: February 17, 2000
Participants on Modification

- online Session: **Alberto Morán**
- offline Session: **Bharath Krishnan, Alan Ng**

<ins>**References and Links**</ins>
(All references are stored at http://collaborate.mit.edu/1.120.html)

• Requirements Specification Version 1.2 for ieCollab Version 1	January 22, 2000
• Requirements Specification Version 1.0 for ieCollab Version 2	December 9, 1999
• Requirements Specification Version 1.1 for ieCollab Version 2	December 22, 1999
• Requirements Specification Version 1.2 for ieCollab Version 2	January 17, 2000
• Requirements Specification Version 1.3 for ieCollab Version 2	January 17, 2000
• Requirements Specification Version 1.4 for ieCollab Version 2	January 18, 2000
• Requirements Specification Version 1.5 for ieCollab Version 2	February 2, 2000

Outline

---

1. Introduction

This draft presents the system requirements of ieCollab version 2. A separate document describes the system requirements for ieCollab Version1 [1]. This document specifies the functional requirements of ieCollab version 2 and how it relates to version 1 system requirements. The purpose of this document is to solicit input from other members of project to ensure that all parties agree on the system requirements.

1.1 ieCollab Versioning System

The ieCollab system will be developed in several phases/versions. Each version focuses on a specific functional component of the overall ieCollab system. The four versions/phases identified are:

- Meeting Management Environment (ieCollab Version 1):
  Allows distributed users to setup and manage online meetings. This includes functionality to keep track of meeting agenda, meeting participants, user profiles and meeting styles.

- Transaction Management (ieCollab Version 2):
  Allows the ieCollab server to track users' usage of ieCollab's meeting management services and charge fees on a per-transaction basis. Allows other A.S.P.'s to provide meeting management services to their clients transparently.

- Collaboration Server (ieCollab Version 3):
  Supports a set of interactive collaboration tools, such as chat tools, whiteboards, for group communications. Users should be able to pay for the use of these collaboration tools on a per-use basis as defined in the transaction management environment fromVersion 2.

- Application Server (ieCollab Version 4):
  Allows multiple distributed meeting participants to work on the same documents concurrently using third-party applications, such as CAD tools and spreadsheet applications. Users should be able to pay for these third-party software on a per-use basis.

This document specifies the system requirements for ieCollab version 2, focusing on the Transaction Management component of the system.

2. General Architecture

ieCollab version 2 will build upon the meeting management capabilities of the version 1. In addition, this version will have the capability to function as a transaction based meeting server. The architecture of which is briefly described below in Figure 1.

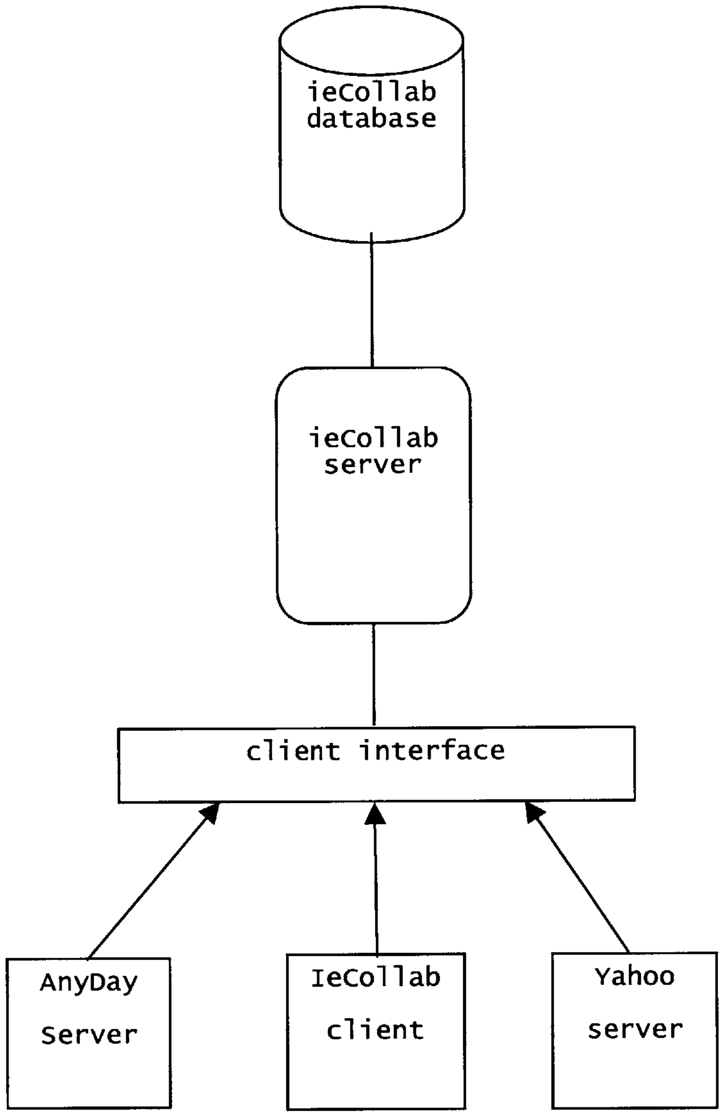


Figure 1: Architecture of ieCollab version 2

ieCollab will provide transparent meeting management services for other Application Service Providers. Take for example a web calendar service like www.Anyday.com [2], which provides basic calendar and scheduling services on the web. ieCollab will offer to handle all the meeting management services for this A.S.P. in a transparent manner. Thus, the thin client interface to ieCollab is replaced by the other A.S.P's web interface. When a user logs on to a meeting through the A.S.P's portal, he will actually be connected to the ieCollab meeting server. This service is charged on a transaction basis. The ieCollab server will keep track of usage of its services. Users will then be billed for the time & applications they used.

3. System Specification

3.1 Overview

The use cases in this document focus on the transaction management component of the system. It is important to note that the use cases presented in this document focus on the use cases of ieCollab as a backend server, which provides transparent meeting management services to end users through another A.S.P. server, such as AnyDay.com or Yahoo.com [3]. The use cases in this document add to the use cases in the ieCollab Version 1 specification. The ieCollab will also be available through ieCollab's own thin client at the same time.

ieCollab Version 2 has the following four functional components (in addition to the functionality in Version 1):

- Set Service Pricing: allows the ieCollab Manager to define pricing schemes for ieCollab services.
- Create A.S.P. Account: allows the ieCollab Manager to establish an A.S.P. account with the ieCollab server
- Create proxy account: allows an A.S.P. to establish an account on ieCollab on behalf of the A.S.P.'s end users transparently; i.e., the personal information of the user present on the A.S.P. is used by the A.S.P. to create the account.
- Track Service Usage: allows the ieCollab server to track usage of ieCollab services on a per-transaction basis (please see use case description on transaction management for more details).

Steps in use cases are grouped into two categories: steps in A.S.P. Server and steps in ieCollab server. Only steps in the ieCollab Server category are considered part of the ieCollab system. Although the A.S.P. Server is an external entity, we specify its corresponding steps in each use case to show how the end user's requests are mapped to ieCollab service requests through the A.S.P. server.

3.2 ieCollab System Entities

To facilitate explanation of the use cases, we describe the following ieCollab system entities and their relationships:

3.2.1 User: a user can be either a Normal User (as mentioned in Version 1 specification [1]) or an A.S.P. User who uses ieCollab via an A.S.P. server. Each user who has an ieCollab User Account is identified by a unique user login name. If the user opts for meeting services through an A.S.P., the A.S.P. will use the user's login name and other data in its database to create the ieCollab account for that user. A user has the following attributes:

- full name
- login name (unique)
- password
- broker identifier
- a list of workgroups the user is in
- a list of scheduled meetings

- a list of preferred meeting templates
- an account profile

The broker identifier specifies the A.S.P. server (such as Yahoo.com or AnyDay.COM) through which the user is using ieCollab. Each user has only one broker. The broker of a user has the right to retrieve and store user profile and meeting setup on behalf of the user.

3.2.2 A.S.P. Server: an A.S.P. Server who has an ieCollab A.S.P. Server Account is identified by a unique server login name. This account is used by the A.S.P. server manager. An A.S.P. server account has the following attributes:

- an A.S.P. name
- login name (unique)
- password
- an account profile

3.2.3 Account Profile: each account profile has the following:

- an account owner
- a pricing agreement
- payment method (usually credit card, could also be a monthly payment)
- billing address
- a list of transactions and invoices
- The GUI definition (When we allow each user a customizable user interface)

The account owner is either a User or an A.S.P. server. (i.e. the A.S.P. Server manager)

3.2.4 Meeting: each meeting has the following:

- a unique meeting ID
- a scheduled date/time
- default meeting template
- agenda
- security level
- A work group (As defined in the Version 1 specification [1])

For more details, please see version 1 specification [1].

3.2.5 Workgroup: A workgroup consists of a list of users and their roles in the group. Multiple meetings can use the same workgroup to define its participant list. Users of a workgroup are also called Workgroup Members. Each workgroup can have one Workgroup Leader and has a unique workgroup ID. For more details, please see section 2.4.2 of the Version 1 specification [1].

3.2.6 Transaction: for each transaction, ieCollab server will record the following:

- date/time of transaction
- principals involved
- usage

The principal of a transaction is the party that initiates the service request and is responsible for paying for the transaction. For example, if a user uses ieCollab directly without going through a 3rd party A.S.P., the user is recorded as the principal. If a 3rdparty A.S.P. server requests for ieCollab services on behalf of its users, the A.S.P. server can identify itself as the principal of the transaction.

Usage is defined by the types of services and quantities of services used. Quantity of a service can be defined by a combination of the following parameters: time duration of service, frequency of access, bytes transferred, bytes of information stored at the server, number of participants in a meeting. An I.S.P revenue model can be used. This includes a standard monthly subscription fee and an hourly charge. The exact rates for various services we offer in future can be decided by the sales management group.

3.2.7 Meeting Template: please see version 1 specification [1].

3.2.8 Meeting Agenda: please see version 1 specification [1].

3.3 Actors in Use Cases

3.3.1 A.S.P. Server Manager
This is the super-user of the A.S.P. server.

3.3.2 A.S.P. User
This is an end-user who is using the A.S.P. server. Although the user may use the ieCollab services through the A.S.P. server, the use of the ieCollab server is transparent to the end user.

3.3.3 ieCollab Manager
This is the ieCollab service manager, who sets usage and pricing policies.

3.4 Use Cases

3.4.1 Use Cases for ieCollab Manager

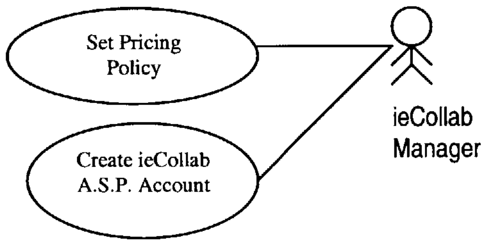


Figure 2: Use cases for ieCollab Manager

3.4.1.1 Setup Pricing Policy

This use case is started by the ieCollab Server Manager. It allows the manager to specify pricing policies for ieCollab services. For each service type, the ieCollab Server Manager can specify the unit definition for measuring service usage, and price per unit. The following service types are provided in Version 2: create/query/update user account and info, create/query/update meetings, create/query/update workgroups, and start/join meetings. As more features are added to ieCollab Version 3 and 4, more service types may be provided.

Service usage maybe measured in terms of:

- time duration of service
- frequency of access
- number of queries performed
- bytes transferred
- bytes of information stored
- number of participants in a meeting.

For each service type, the ieCollab Server Manager must specify which type of measurement will be used, and the price per unit of service usage.

3.4.1.2 Create ieCollab A.S.P. Account

An ieCollab Manager must setup an A.S.P. server account for the A.S.P. with ieCollab before the A.S.P. users can use the ieCollab services. To create an A.S.P. Account, the following information must be provided: name of A.S.P., login name, password, contact name/address, and billing information as specified by the contract between A.S.P. and ieCollab.

3.4.2 Use Cases for A.S.P. User

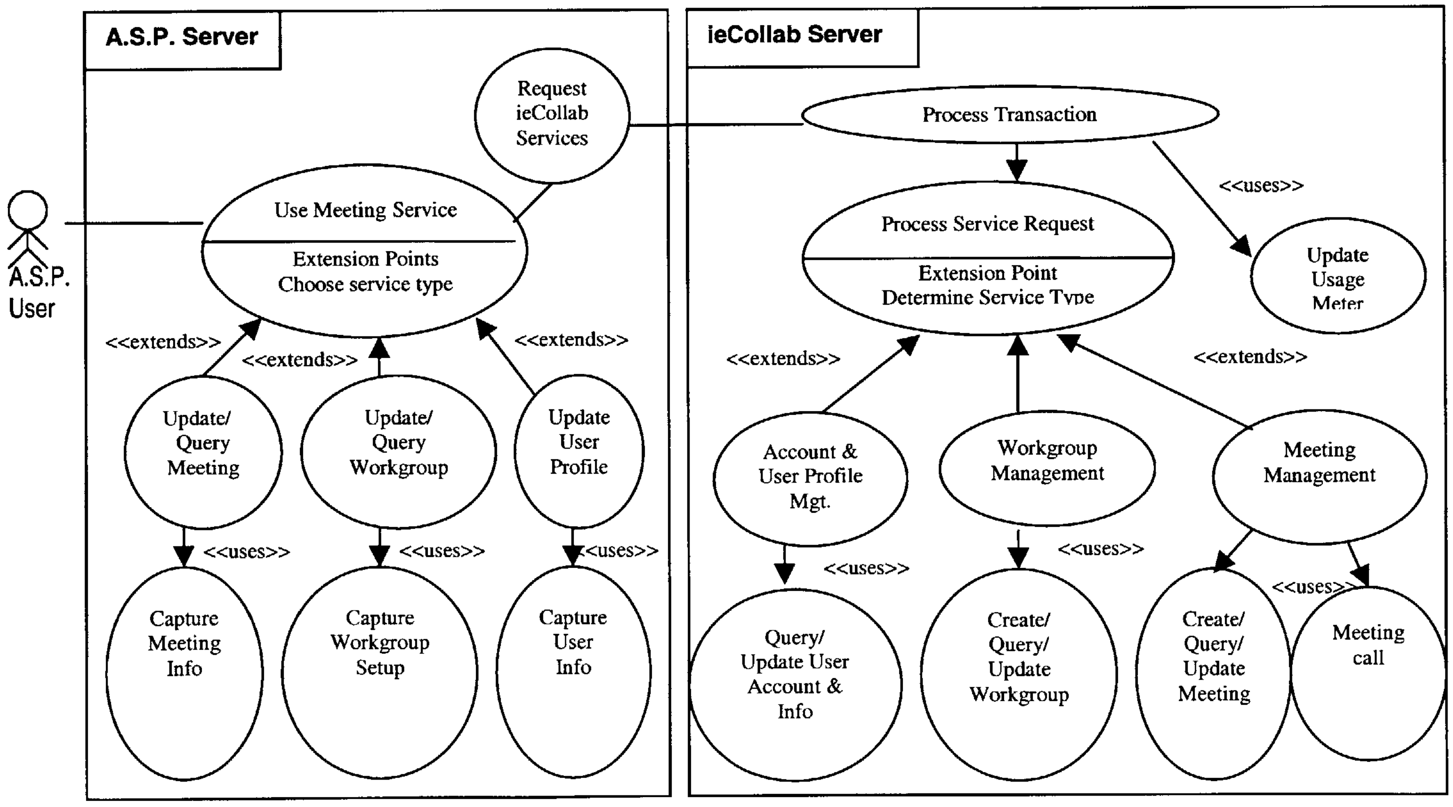


Figure 3a: Use Cases for A.S.P. User (Process Transaction)

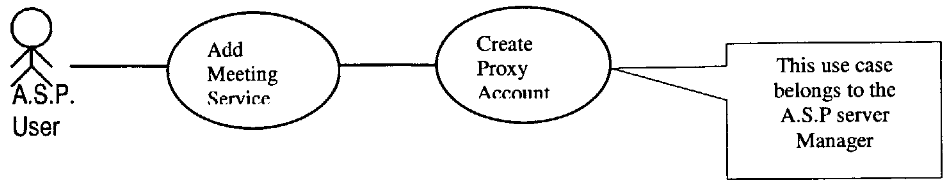


Figure 3b: Use Cases for A.S.P. User (Add Meeting Service)

3.4.2.1 Process Transaction

This use case is started by an A.S.P. User when requesting to use meeting service. The A.S.P. User's request will trigger the A.S.P server to send a service request to the ieCollab server. The service request will contain the A.S.P. user's login name and password, type of service requested, and other necessary data for completing the request. It is to be noted that the A.S.P. server acts on the behalf of the user. Any reference in this section to the A.S.P. server should be construed as the A.S.P. server acting on behalf of the user. The ieCollab server will process the service request as a transaction. The following steps are included in the Process Transaction use case:

- **Process Service Request:** There are several extended use cases based on the service types. The three main categories of sub use cases are: Account and User Profile Management, Workgroup Management, and Meeting Management. These are explained in section 3.4.2.1.1

- **Update Usage Meter:** Logging and tracking an A.S.P. server's or individual A.S.P. user's usage of ieCollab services

3.4.2.1.1 Sub Use Cases in Process Service Request

The following use cases are extended from the Process Service Request use case:

3.4.2.1.1.1 Account and User Profile Management: this use case includes Creating, Updating, and Querying of ieCollab User Accounts and User Profiles. ieCollab allows an external A.S.P. server to perform the following types of requests:

- **Query User Profile:** given a user login name, ieCollab server retrieves and returns that user's user profile.

- **Store User Profile:** given a user login name and a user profile, ieCollab server stores the user profile for that user.

3.4.2.2 Workgroup Management

This use case is triggered when the A.S.P user requests to Create/Modify Workgroup through the A.S.P server. This use case corresponds to the Workgroup Management use case in ieCollab Version 1 specification. Readers should refer to Version 1 specification [1] for details of this use case.
As an interface to its workgroup management, ieCollab allows an external A.S.P. server to perform the following types of requests:

- **Query Workgroup:** query workgroups based on workgroup IDs. This ID is a unique identifier for the workgroup.

- **Store Workgroup:** given a workgroup identifier, store a workgroup setup information. Please see section 3.2 for a detailed specification of a workgroup.

3.4.2.3 Meeting Management

This use case is triggered when the A.S.P user starts a Meeting Call through the A.S.P server. Note that ieCollab server expects the A.S.P server to capture meeting selection. This use case corresponds to the Meeting Management use case in ieCollab Version 1 specification. Readers should refer to Version 1 specification [1] for details of this use case.

ieCollab allows an external A.S.P. server to perform the following types of requests.

- **Query Ongoing Meetings:** this will cause ieCollab server to return a list of ongoing meetings currently at that ieCollab server.

- **Query Meeting:** given a meeting identifier, ieCollab server will return the meeting setup associated with that meeting. Please see section 3.2 for a detailed specification on the attributes of a meeting entity.

- **Store Meeting:** store meeting setup information under the specified meeting identifier. Please see section 3.2 for a detailed specification on the attributes of a meeting entity.

- **Start Meeting:** given a meeting specifier and a user ID, ieCollab will start a meeting using the meeting setup associated with the meeting identifier. All meeting state (current participants and their locations, meeting logs, communication channels among participants) will be maintained at ieCollab server.

- **Join Meeting:** given a meeting specifier and a user ID, ieCollab will connect the user to the specified meeting. An error status will be returned if no on-going meetings correspond to the meeting specifier. An access denied message will be returned if the user ID is not permitted to join that meeting.

3.4.2.4 End Transaction

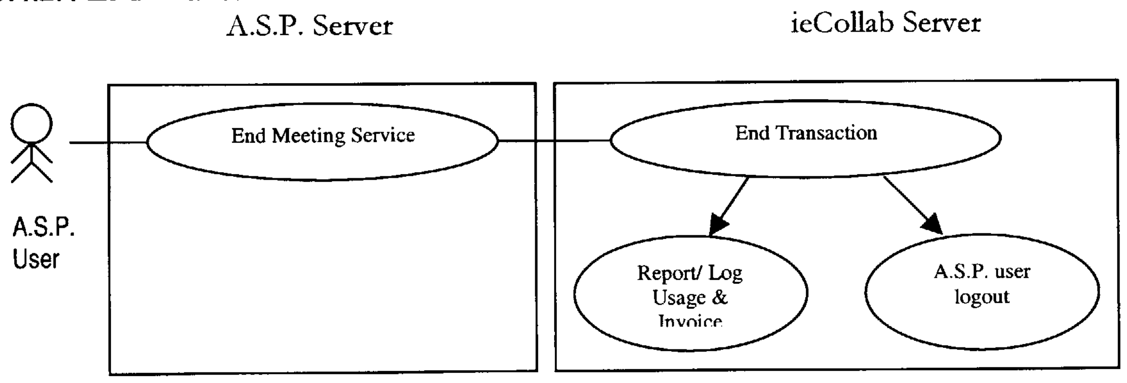


Figure 4: Use Cases for A.S.P. User (End Transaction)

The transaction is considered terminated when the A.S.P. user terminates the use of the ieCollab's meeting service. ieCollab will report the final service usage and invoice back to the A.S.P. server, and the A.S.P. server may decide to logout of the ieCollab server at the end of the transaction.

3.4.3 Use Cases for A.S.P. Server Manager

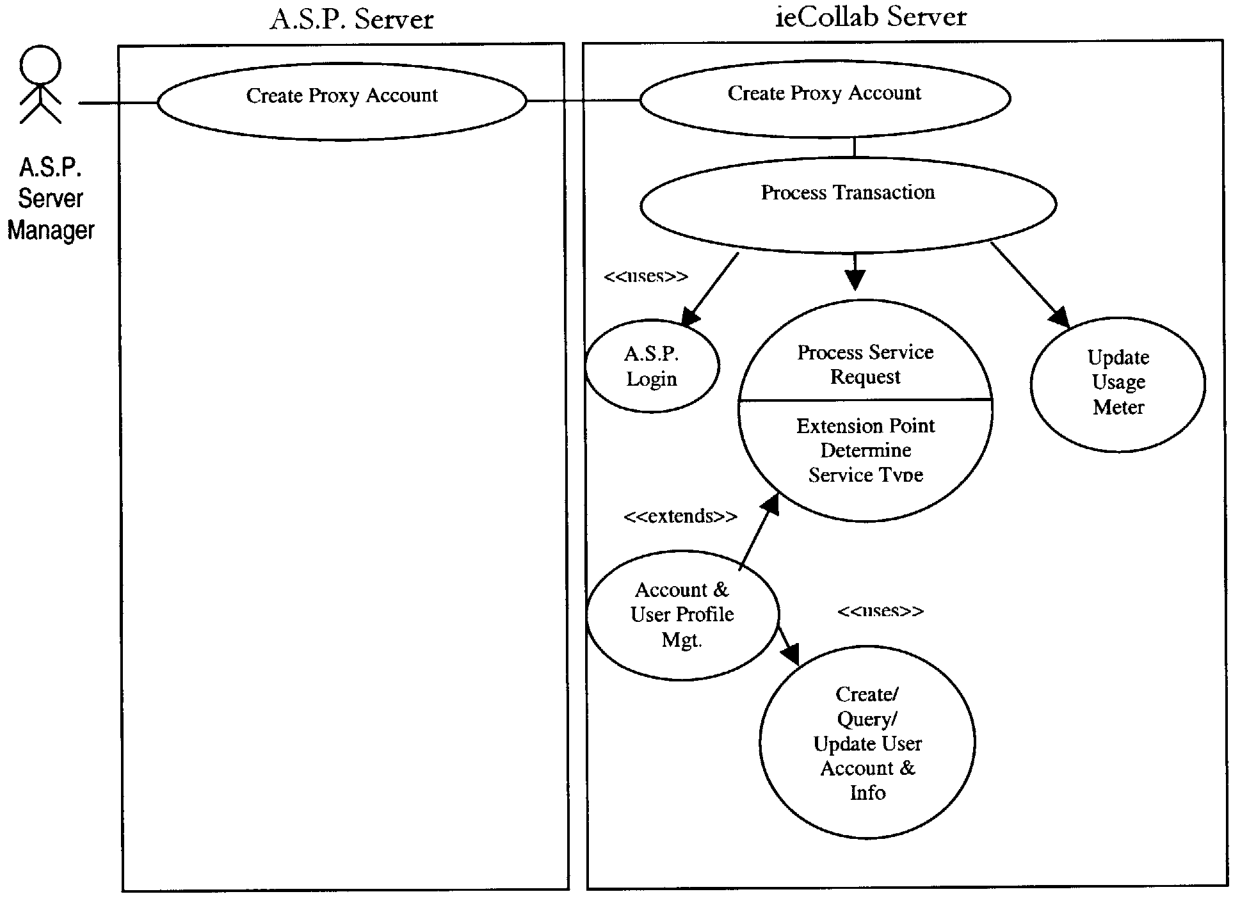


Figure 5: Use Cases for A.S.P. Server Manager (Create Proxy Account)

3.4.3.1 Create a proxy account

This use case is used by the A.S.P. server manager when the A.S.P. user opts for meeting management services. The process of creating an ieCollab account is automated by this use case. In other words, the A.S.P. uses the user profile for the user it has in its database to create an account for the user on the ieCollab server.

3.4.3.2 A.S.P. Login: An A.S.P. server can log itself in the ieCollab server by providing a valid ieCollab A.S.P. account name and password. Once an A.S.P. server logs in, it can request various ieCollab services, such as to create new user accounts, store/retrieve user profiles and meeting setups on behalf of its A.S.P. users. This step applies to A.S.P.'s which have a pre-established ieCollab A.S.P. accounts, which includes the pricing agreement and payment information.

3.4.3.3 Create User Account: An A.S.P. server can create an ieCollab user account on behalf of a user transparently. In order to do so, an A.S.P. server must log itself into ieCollab server first by providing a valid login name and password. After the A.S.P. server has been authenticated, it must then specify the user's name, login name, and selected password. In response, the ieCollab server must then create an ieCollab user account using the given information, and mark the A.S.P. server as the broker of the user.

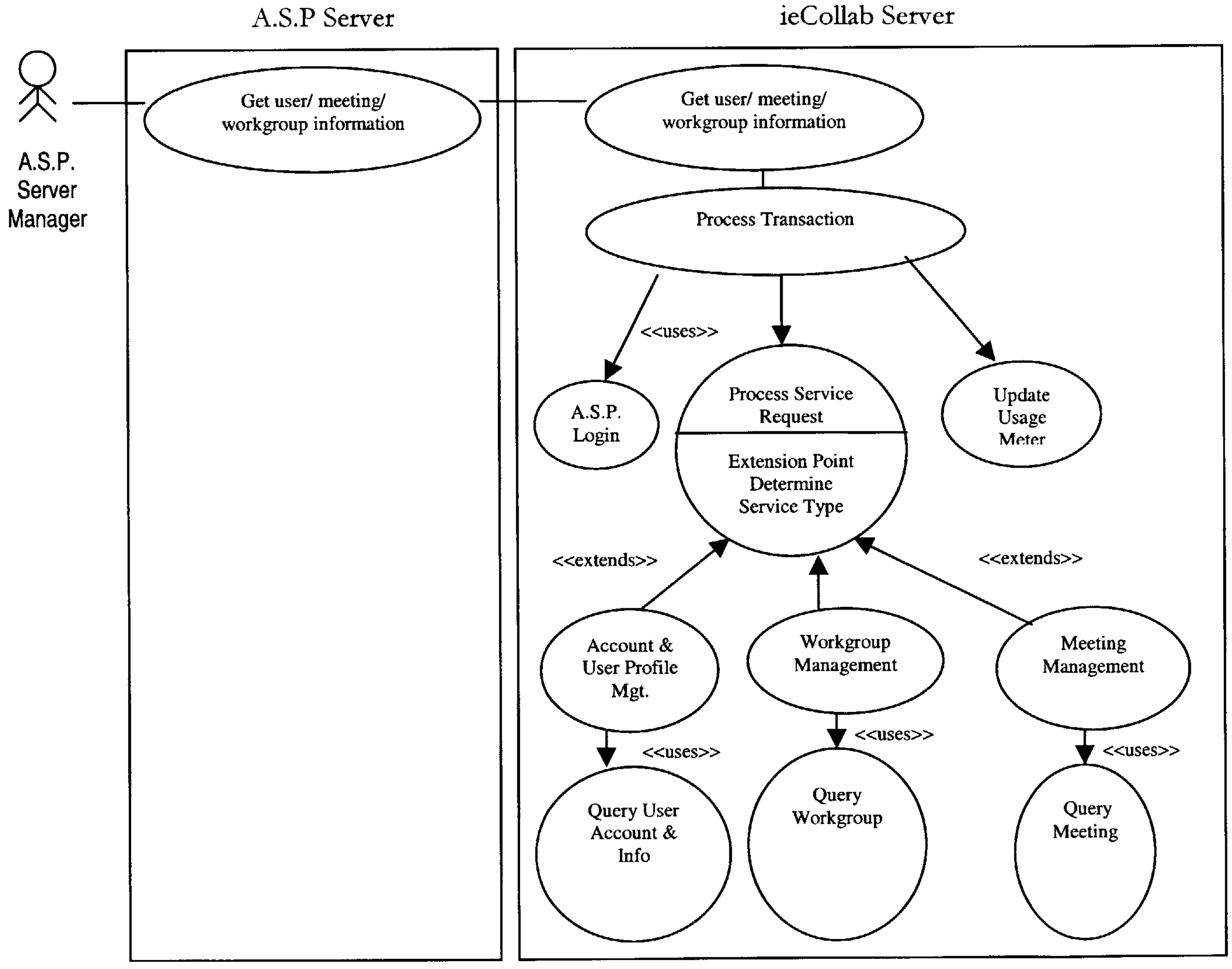


Figure 6: Use Case for A.S.P. Server Manager (Get User/ Workgroup/ Meeting Information)

3.4.3.4 Get user/ workgroup/ meeting information

This use case is used by the A.S.P. server to get information from ieCollab about its users so that its own databases can be updated. An example would be an A.S.P which provides calendar services. The A.S.P Manager would then be able to automate the following task: Get meeting schedule information for users, update the calendar database so that the user's calendar would reflect the scheduled meeting.

References

[1] Moran A. "ieCollab Version 1 specification 1.4". 2000

[2] http://www.AnyDay.com February 3, 2000

[3] http://www.yahoo.com February 3, 2000

Appendix B

ieCollab Transaction Management

Design Specification V 1.0

Updated by: Hao Chen
Date: 3/1/2000
Participants on Modification:

- offline Session
  Wassim Solh, Manuel Alba, Hao Chen, Sugata Sen, Anup Mantena

References and Links

Requirement Specifications for ieCollab Version 1 and 2
IeCollab version 1 Design Document v0.9, February 19,2000
IeCollab version 1 Design Document v0.5, February 10,2000
Design Document v0.2, January 25, 2000
Design Document v0.1, December 1, 1999

Outline

1. Introduction

1.1. Purpose

The purpose of this document is to describe the methodology for the implementation of basic architecture and the Transaction Management of ieCollab as specified by the final Requirements Analysis documentation.

1.2. Scope

This document presents the design of the basic architecture, account management, user management, and transaction management tools of ieCollab version 1. It identifies the classes to implement and interactions between classes. The design of the ieCollab system is based on three-tier design concept, and Corba technology (see glossary).

Separate documents present the specifications for:

- Meeting Management (Version 2)
- Collaboration Server (Version 3)
- Application Server (Version 4)

1.3. Modifications

Major modifications from the previous version include:

1) Deleted of DBInterface class so that all the objects can interact with database directly and reduced bottleneck.
2) Added Corba service to ClientWindow so that the server can call back to some functions on the client.
3) Moved some methods from CollabUser to Workgroup and Meeting so that client can interact directly with those objects.
4) Integrated database diagram to this file.
5) Provided IDL for client/server interface definitions.

2. System Architecture

The ieCollab system comprises of three layers: user services (front end user interface), business services (business logics), and data services (back end database). The fist layer is linked to the second layer through Corba connection and the second layer is connected to database via JDBC.

Figure 2 shows the basic system objects of ieCollab. On the user layer, an object of CollabClient class is responsible for basic UI functions. CollabClient opens other dialogs or windows to display information and get user inputs.

On the server side, before a client logs into the ieCollab system, a CollabServer object takes care of the interactions between the client and our server. After a user logs in to the system, a CollabUser object will be created for the user. Then, the client can require some general services and user account services through this CollabUser object. If the user needs services related to Workgroup and Meeting objects, references to specific Workgroup and Meeting object can be obtained by using getWorgroupRef(ID) and getMeetingRef(ID) and providing WorkgroupID or MeetingID as a parameter. After the client side gets the reference to the Workgroup or Meeting object, the client can use the methods provided by that object, such as getMemberList(), etc.

All the database operations are done by individual methods. Each method that needs information from database opens connection to the database and does appropriate queries to retrieve data. After the database operation is done, connection to database should be closed. Some of the complex database operations can be put on the database by using stored procedures. The use of stored procedure can improve database performance.

Figure 1. Layers and Connections of the system

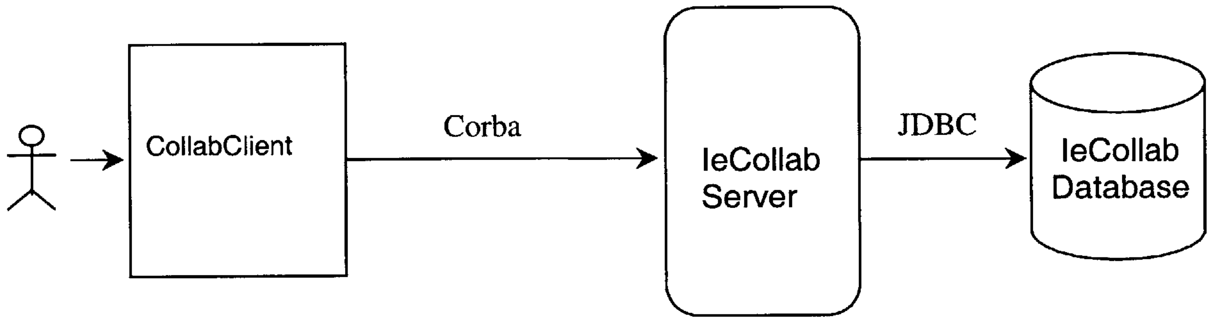
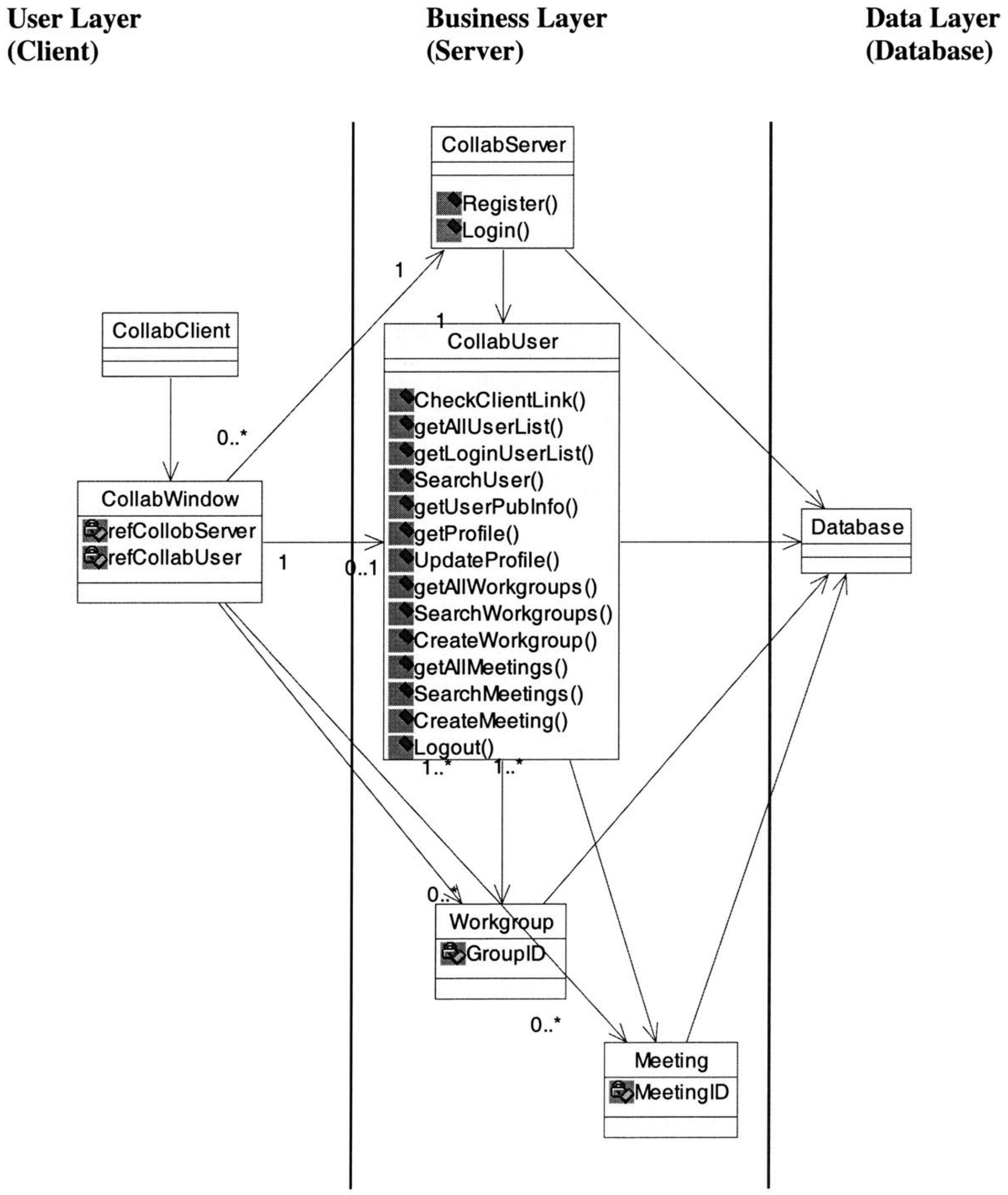
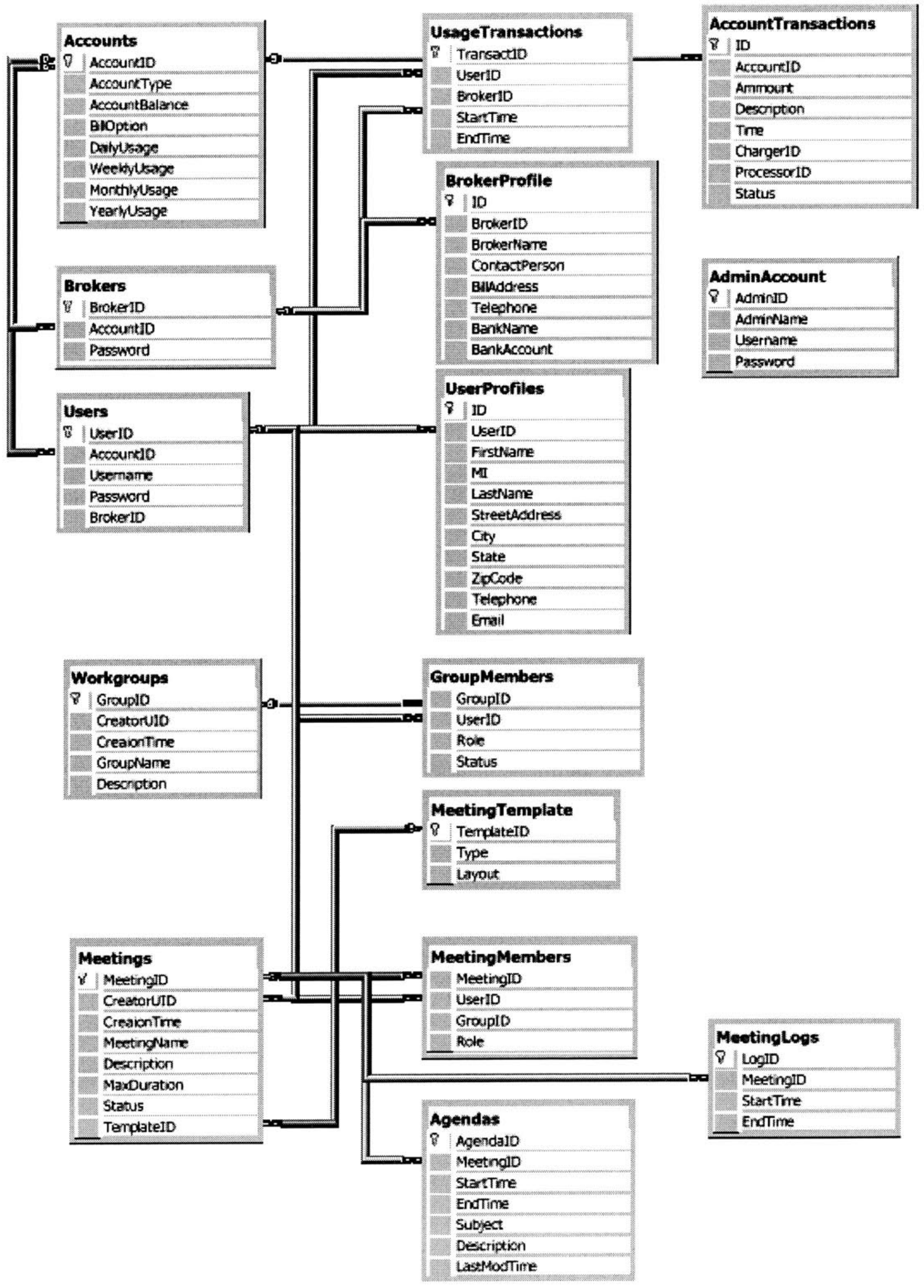


Figure 2. System Architecture



3. Database Architecture

Figure 3. Database Diagram



1-1

4. Program Architecture

4.1. Client / Server Interface

The following IDL file defines the interfaces for client/server interaction

```
module CollabApp
{
   interface CollabClient;
   interface CollabServer;
   interface CollabUser;
   interface Workgroup;
   interface Meeting;

   //******** Basic Data Structures *********
   struct ListItem
  {
         long ID;
         string Name;
   };

   struct UserPubInfo
  {
         long UserID;
         string UserName;
         string FirstName;
         string LastName;
         string City;
         string State;
  };

  struct UserProfile
 {
         string FirstName;
         string MI;
         string LastName;
         string StreetAddress;
         string City;
         string State;
         string ZipCode;
         string Telephone;
         string Email;
   };

  struct BrokerProfile
  {
        string BrokerName;
        string ContactPerson;
        string BillAddress;
        string Telephone;
        string BankName;
        string BankAccount;
};

 typedef sequence<string> namelist;
 typedef sequence<long> idlist;
typedef sequence<ListItem> displist;

// ****** Interfaces ********

interface CollabClient
{
        string sayHello();

        void RefreshWGlist();
        void RefreshMGlist();
        void RefreshHMGlist();
        void RefreshInvitedMGlist();
};


interface CollabServer
{
        string sayHello();

        long Register(in string Username, in string Password,
                        in UserProfile Profile);

        CollabUser Login(in string Username, in string Password,
                        in CollabClient refClient);

        CollabUser LoginBroker(in string Username, in string Password,
                        in long BrokerID, in string BrokerPassword,
                        in CollabClient refClient);
};


interface CollabUser
{
        string getID();
        string getUsername();
        displist getMyWorkgroupList();
        displist getMyMeetingList();
        displist getMyHistoryMeetingList() ;
        displist getMyInvitedMeetingList();
        displist getAllUserList();
        displist getLoginUserList();
        displist SearchUsers(in string Searchstring);
        UserPubInfo getUserPubInfo(in long UserID);
        UserProfile getProfile();
        long UpdateProfile(in UserProfile Profile);
        boolean Logout();

        displist getAllWorkgroups();
        displist SearchWorkgroups(in string str);
        Workgroup getWorkgroupRef(in long WorkgroupID);
        long CreateWorkgroup(in string WorkgroupName, in string Description);
        long DeleteWorkgroup(in long WorkgroupID);

        displist getAllMeetings();
        displist SearchMeeting(in string str);
        Meeting getMeetingRef(in long MeetingID);
        long CreateMeeting(in string MeetingName, in string Description);
        long DeleteMeeting(in long MeetingID);
  };

};
```

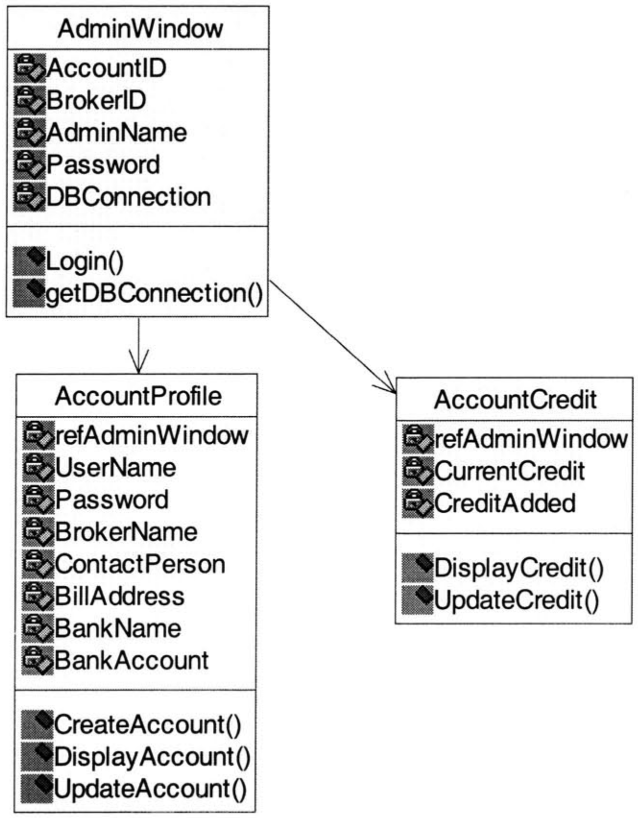
4.2. Basic Data Structures Mapped to Java

```
Class ListItem
{
        int ID;
        String Name;
};

class UserPubInfo                // the public information of a user
{
        long UserID;
        String UserName;
        String FirstName;
        String LastName;
        String City;
        String State;
}

class UserProfile
{
        String FirstName;
        String MI;
        String LastName;
        String StreetAddress;
        String City;
        String State;
        String ZipCode;
        String Telephone;
        String Email;
}

class BrokerProfile
{
        String BrokerName;
        String ContactPerson;
        String BillAddress;
        String Telephone;
        String BankName;
        String BankAccount;
}
```

4.3. Administration Classes

The administration tool is a separate tool for system administrators to create new broker accounts, edit accounts, and add credits to accounts. The administration tool is for internal use and will not be distributed to clients.

Figure 4. Classes for Administration Tools



```
class AdminWindow
{
        //********* Atributes ****************
        private long AccountID;  // ID of Current Account
        private long BrokerID;
        private String AdminName;
        private String Password;
        private Connection DBConnection ;

        //********* Constructor ****************
        public AdminWindow()
        {
                // Show a Login Dialog
        }

        public Boolean Login(String username, String Password)
        {
                // check login information
                //  setup DBConnection to database if login succeed
        }

        public getDBConnection ()
        {
                // return this.DBConnection
        }

}

class AccountProfile
{
        //********* Atributes ****************
        private AdminWindow refAdminWindow;

        private String UserName;  // ID of Current Account
        private String Password;
        ....

        public int CreateAccount()
        {
                // Use current display information to create an account
                // return error code
        }

        public void DisplayAccunt(AccountID)
        {
                // refAdminWindow.getDBConnection() and query database
                // Display in the window
        }

        pubic int UpdateAccount (String Username, String Password, ....)
        {
                // Write Account Information to database
                // Retrun Error Code
        }

}


class AccountCredit
{
        //********* Atributes ****************
        private AdminWindow refAdminWindow;

        private float CurrentCredit;
        private float CreditAdded;
        ....

        public void DisplayCredit(AccountID)
        {
                // AdminWindow.getDBConnection and query database
                // Display in the window
        }

        public int UpdateCredit()
        {
                // Confirm()
                // Writeto Database
                // return error code
        }

        private boolean Confirm()
        {
                // Retrun 'true' if confirmed by a dialog
        }

}
```

4.4. CollabClient Classes

See user interface definition document.

Figure 5. CollabServer and CollabUser Classes

```
CollabServer
--------------------------------------------------
private static ORB orb
--------------------------------------------------
public CollabServer()

public ORB getORB() {return this.orb;}
public String sayHello()

public long Register(String Username, String
Password, UserProfile Profile)

public CollabUser Login(String Username, String
Password, String strCollabWindow)

public CollabUser Login(String Username, String
Password, long BrokerID, String BrokerPassword,
String strCollabWindow)

private void StartTimer()

private void CheckUsers()
```

```
CollabUser
--------------------------------------------------
private static ListItem[] LoginUIDList;
private static HashMap UserRefMap;

private long UserID;
private String Username;
private long BrokerID;
private long TransactionID;
private CollabWindow refCollabWindow;

private ListItem[] myWorkgroupList;
private ListItem[] myMeetingList;
private ListItem[] myHistoryMeetingList;
private ListItem[] myInvitedMeetingList;
--------------------------------------------------
public CollabUser(long UserID, long BrokerID, String
strCollabWindow)
public CollabUser(long UserID, long BrokerID, String
strCollabWindow)

public String getID() {return this.UserID;}
public String getUsername() {return this.Username;}
public ListItem[] getMyWorkgroupList()
public ListItem[] getMyMeetingList()
public ListItem[] getMyHistoryMeetingList()
public ListItem[] getMyInvitedMeetingList()

static public CollabUser getUserRef(long UserID)
public ListItem[] getAllUserList() {
public ListItem[] getLoginUserList() {
public ListItem[] SearchUsers(String str)
public UserPubInfo getUserPubInfo(long UserID)
public UserProfile getProfile()
public long UpdateProfile(UserProfile Profile)
public boolean Logout()

public ListItem[] getAllWorkgroups()
public ListItem[] SearchWorkgroups(String str)
public Workgroup getWorkgroupRef(long WorkgroupID)
public long CreateWorkgroup(String
WorkgroupName,String Description)
public long DeleteWorkgroup(long WorkgroupID)

public ListItem[] getAllMeetings()
public ListItem[] SearchMeeting(String str)
public Meeting getMeetingRef(long MeetingID)
public long CreateMeeting(String MeetingName, String
Description)
public long DeleteMeeting(long MeetingID)

private boolean ResolveWindow(String str)
```

4.5. CollabServer Class

```
class CollabServer
{
        //********* Atributes ****************
        private static ORB orb;

        //********* Constructor ****************
        public CollabServer()
        {
                // set up all attributes; StartTimer()
        }

        //********* Attribute Access ****************
        // For server use, Don't put this method in IDL
        public ORB getORB() {return this.orb;}

        //********* Public Methods ****************

        public String sayHello()
        {
                return "Hello!";
        }

        public long Register(
                String Username, String Password, UserProfile Profile)
        {
                // check and put info. to Account, Users, User Profile tables
                // return a new UserID;
        }

        public CollabUser Login(String Username, String Password, CollabClient refCollabClient)
        {
                // Check login information and create CollabUser object
                // return a reference to a CollabUser object if login succeeds.
        }

        public CollabUser LoginBroker(String Username, String Password,
                                long BrokerID, String BrokerPassword, , CollabClient refCollabClient)
        {
                // Brokers use this method by passing the broker's login info.
                // Check login information
                // create CollabUser object
                // using CollabUser(long UserID, long BrokerID, CollabClient refCollabClient)
                // return a reference to a CollabUser object if login succeeds.
        }

        //********* Private Methods ****************
        private void StartTimer()
        {
                // Start a timer that calls the CheckUsers method regularly
                // to find out whether the users are still linked.
        }

        private void CheckUsers()
        {
                // go through CollbUser.LoginUserList and call CheckClientLink
                // of each CollabUser object
        }

}
```

4.6. CollabUser Class

```
class CollabUser
{
        private static ListItem[] LoginUserList; // Store the list of all login users' IDs and names.
        private static HashMap UserRefMap;  // Map UserID to refCollabUser if the user is online

        //********* Atributes *************
        private long UserID;
        private String Username;
        private long BrokerID;
        private long TransactionID;                // ID of current transaction
        private CollabClient refCollabClient; // reference to client's window in order to callback

        private ListItem[] myWorkgroupList;     // A list of workgroup IDs and names the user is in
        private ListItem[] myMeetingList;        // A list of meeting IDs and names the user is in
        private ListItem[] myHistoryMeetingList;// A list of meeting IDs and names the user has been in
        private ListItem[] myInvitedMeetingList;// meeting IDs and names the user has been invited

        //********* Attribute Access ****************
        public String getID() {return this.UserID;}
        public String getUsername() {return this.Username;}

        public ListItem[] getMyWorkgroupList() {return this.myWorkgroupList;}
        public ListItem[] getMyMeetingList() {return this.myMeetingList;}
        public ListItem[] getMyHistoryMeetingList() {return this.myHistoryMeetingList;}
        public ListItem[] getMyInvitedMeetingList() {return this.myInvitedMeetingList;}

        //********* Constructor ****************
        public CollabUser(long UserID, long BrokerID, CollabClient refCollabClient)
        {
                // Initialize object
                // Create new transaction
                // set refCollabClient
        // set up all attributes:
}

//********* Public Methods ****************

public boolean CheckClientLink()
{
        // call refCollabClient.sayHello() to see if the client is still there.
        // return 'true' if still linked.
}

// For server use, not in IDL

public static CollabUser getUserRef(long UserID) {
        return CollabUser.UserRefMap.get(UserID);
}

public static ListItem[] static_getLoginUserList() {
        return CollabUser.LoginUserList;
}

public static ListItem[] static_getAllUserList() {
        // Query database and return a list of all users registered;
}

public static ListItem[] static_SearchUsers(String str) {
        // Query database and return a list of users that meet the search String;
}

// Implementations for IDL

public ListItem[] getLoginUserList() {
        return static_getLoginUserList();
}

public ListItem[] getAllUserList() {
        return static_getAllUserList();
}

public ListItem[] getLoginUserList() {
        return CollabUser.static_getLoginUserList;
}

public ListItem[] SearchUsers(String str) {
        return CollabUser.static_ SearchUsers(String str);
}

public UserPubInfo getUserPubInfo(long UserID) {
        // Query database or use reference to an online user to
        // return UserPubInfo for a specific user;
```

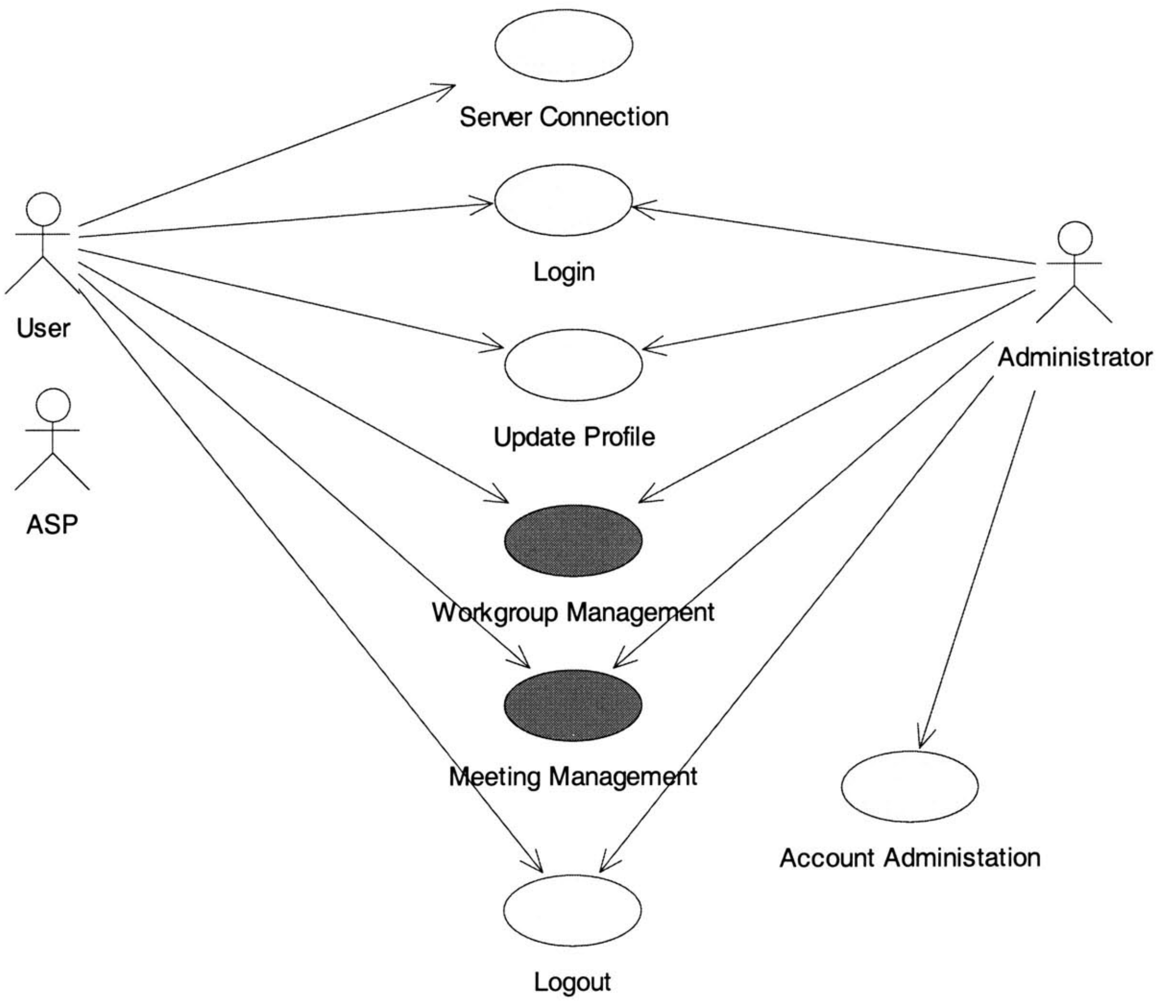
```
}


//********* Methods for User Management ****************

public UserProfile getProfile()
{
        // return the user's Profile
}

public long UpdateProfile(UserProfile Profile)
{
        // put new Profile info into database
        // return error code.
}

public boolean Logout()
{
        // Update transaction logout time, user's usage info, and broker's usage info.
        // remove user from LoginUserList and disconnect ORB.
        // return 'true' if succeed.
}


//********* Methods for Workgroup Management ***************

public ListItem[] getAllWorkgroups()
{
        // return Workgroup.allWorkgroupList
}

public ListItem[] SearchWorkgroups(String str)
{
        // return workgroups that contain str in description
}

public Workgroup getWorkgroupRef(long WorkgroupID)
{
        // return a Workgroup reference
}

public long CreateWorkgroup(String WorkgroupName,String Description)
{
        // a user creates a new work group
        // return error code
}

public long DeleteWorkgroup(long WorkgroupID)
{
        // call Workgroup.DeleteWG(this,WorkgroupID)
```

```
            // return error code
        }

        //********* Methods for Meeting Management ****************

        public ListItem[] getAllMeetings()
        {
            // return Meeting.allMeetingList
        }

        public ListItem[] SearchMeeting(String str)
        {
            // return meetings that contain str in description
        }

        public Meeting getMeetingRef(long MeetingID)
        {
            // return a Meeting reference
        }

        public long CreateMeeting(String MeetingName,String Description)
        {
            // return error code
        }

        public long DeleteMeeting(long MeetingID)
        {
            // call Meeting.DeleteMG(this,WorkgroupID)
            // return error code
        }


        //********* Private Methods ****************
        private boolean ResolveWindow(String str)
        {
            // Destringify the str and get the reference to CollabClient.
            // return 'true' if succeeded.
        }

        private boolean DisconnectORB()
        {
            // disconnect this object with orb so that system can garbage collect it.
            // return 'true' if succeeded.
        }
}
```

5. Use Cases and Sequence Diagrams

Use cases for ieCollab include server connection, login, update profile, workgroup management, meeting management, logout, and account administration. Design for workgroup and meeting management is a large part and is included in a separate document.

Figure 6. Main Use Cases



5.1. Account Administration

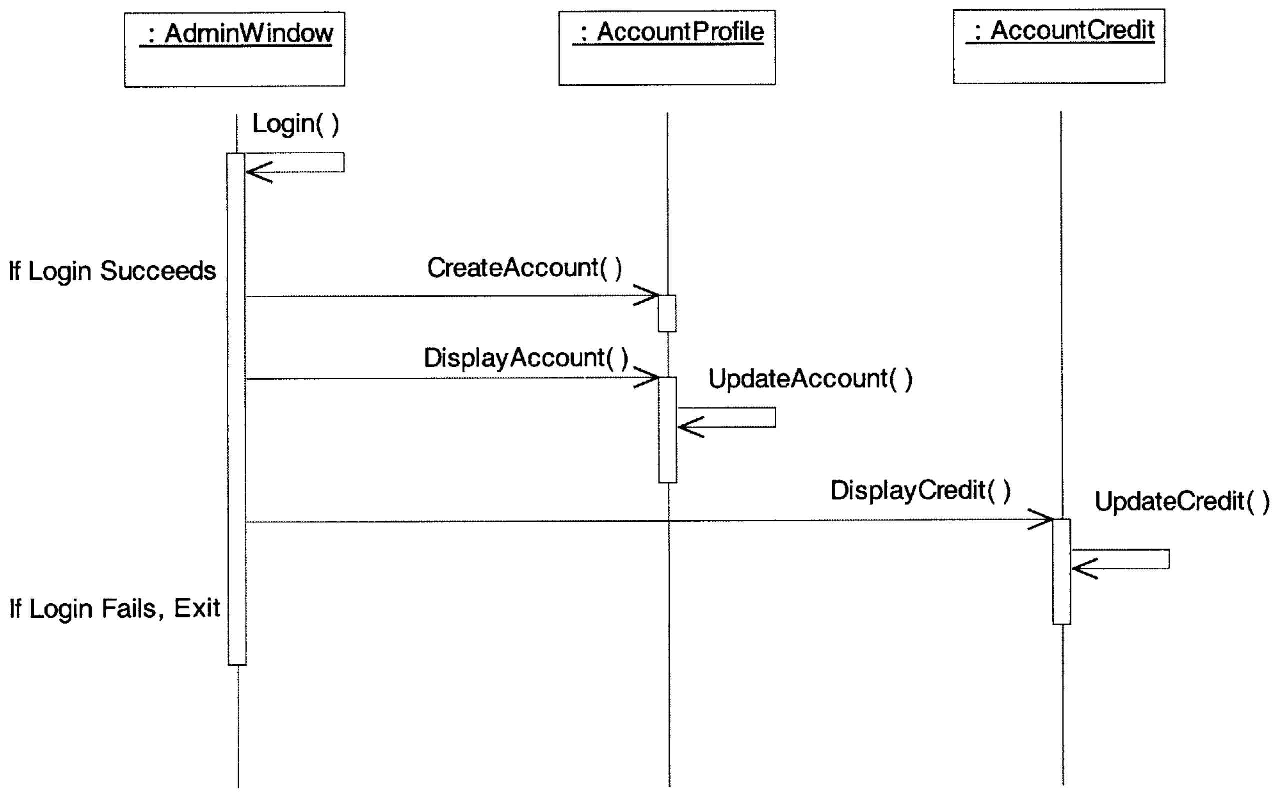
System administrator creates records in the database for a new broker or updates account for a broker. Need to implement with user interface.

Create Account

Update Account

Add Credit

Figure 7. Account Administration Diagram



5.2. Server Connection

When a CollabClient object is started on the client side, it should resolve the reference to the CollabServer object. This is done by a Corba reference resolution. We recommend use stringified object resolution (see demo code) if we are going to implement the client UI with applets.

The stringify/destringify process is: 1) when the server object is started, it create a string that represents the object. 2) the string is passed to the client applet using parameters. 3) the client gets the object reference by transforming the string back to object.

5.3. *User Management:*

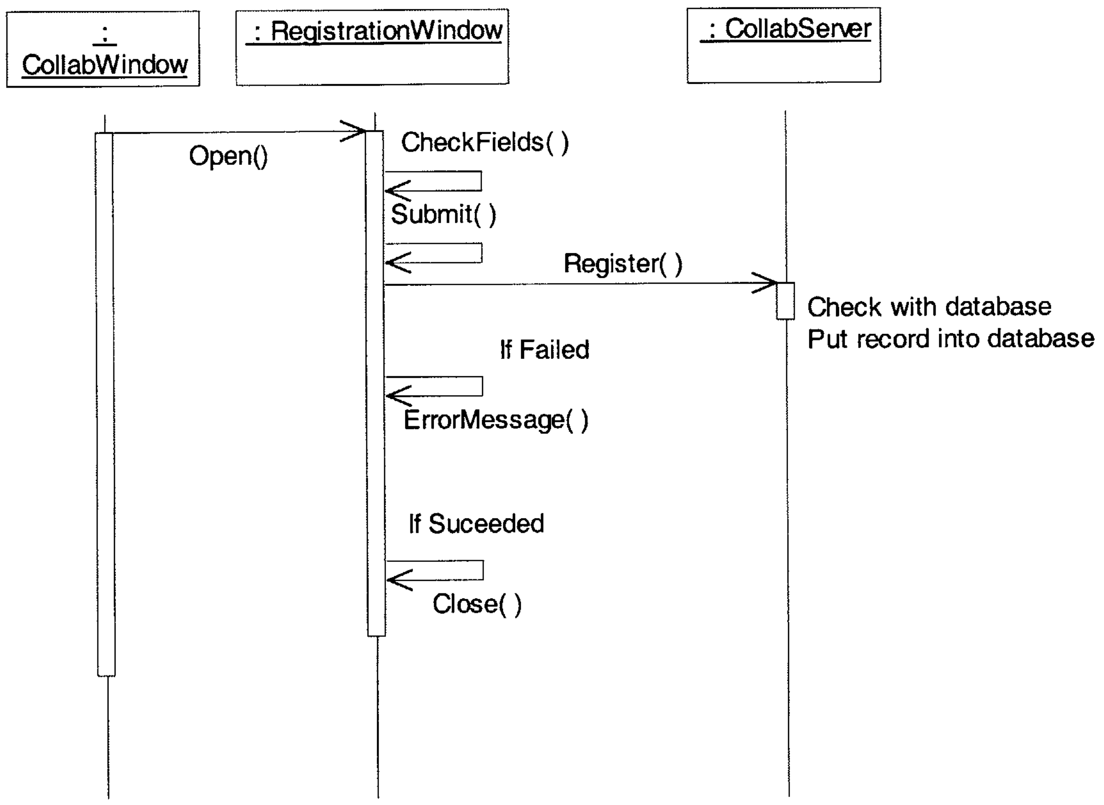
Registration

A user opens a registration window and fills in the information needed. When the user presses the registration button, the information is first checked to make sure that basic information provided and the confirmation password matched. Then, the information is send to the register method of the CollabServer object. CollabServer will check the information the user provided with the database. Success is returned when the registration go through the check and recording to the database. Failure is returned then username is occupied or there is another problem.

User Profile update

This use case is very similar to registration. The client window need to first get the existing user profile to the client side and then send the information to CollabUser:UpdateProfile().

Figure 8. Registration Sequence Diagram



5.4. Transaction Management:

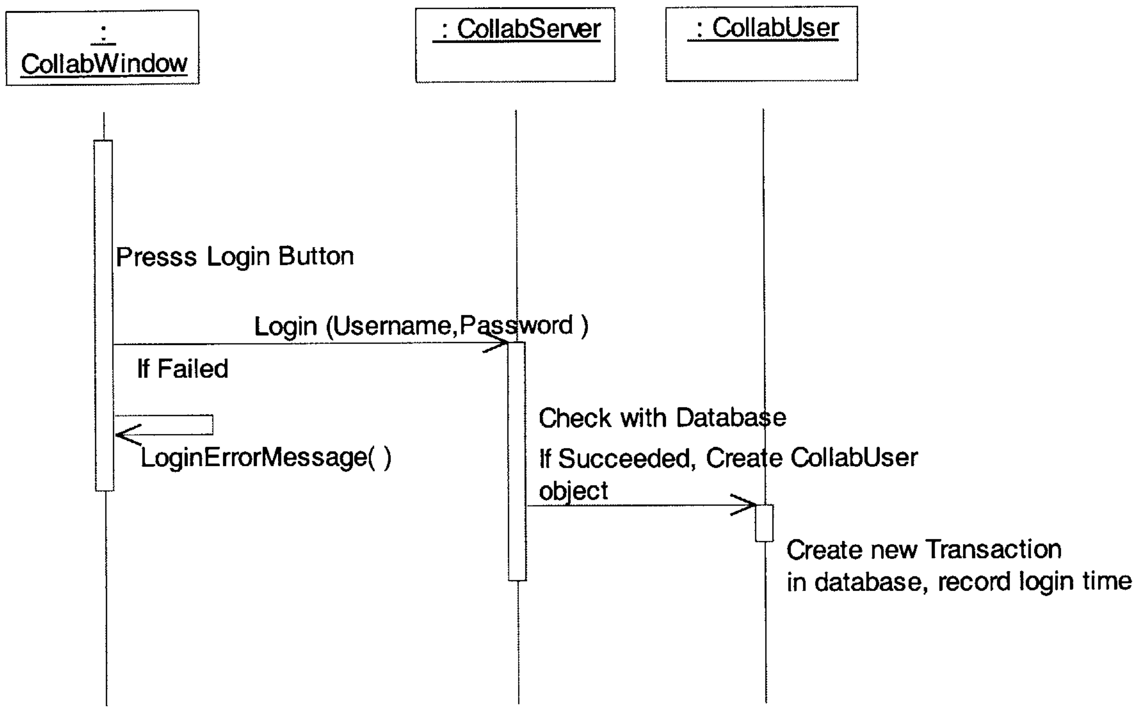
Login for individual users

When a user press login button, CollabClient sends the username and password information to the CollabServer object. CollabServer in turn checks the database to see if the login information is correct. If the login failed, an error message is returned. Otherwise, CollabServer creates a CollabUser object for the user and CollabUser provides all the interfaces for the user to further interact with the ieCollab system; at the same time, the constructor of CollabUser creates a new transaction record in the database.

Login for users through brokers

When a user login through brokers, the broker application sends the username and password information of the user and the BrokerID and BrokerPassword information to the CollabServer object. CollabServer in turn checks the database to see if the login information of both the user and the broker is correct. If the login failed, an error message is returned. Otherwise, CollabServer creates a CollabUser object for the user; at the same time, the constructor of CollabUser creates a new transaction record in the database.

Figure 9. Login Sequence Diagram

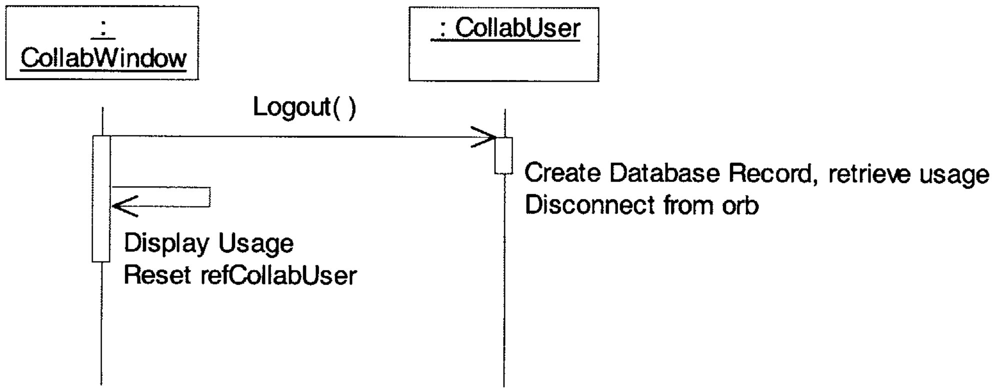


: CollabWindow
: CollabServer
: CollabUser
Presss Login Button
Login (Username,Password)
If Failed
LoginErrorMessage( )
Check with Database
If Succeeded, Create CollabUser object
Create new Transaction in database, record login time

Logout

When the user press logout button, CollabClient or a broker application calls CollabUser to log out. CollabUser then update database: 1) record transaction end time. 2) calculate usage for this transaction. 3) record usage in user's account. 4) if necessary, record usage to the broker's account. When the database is updated, the CollabUser object should disconnect with the system ORB so that the garbage collector can delete this object. Usage information will be returned to the client window and displayed. The refCollabUser on the client side should also be reset.

Figure 10. Logout Sequence Diagram



6. Conclusion

This design document describe the implementation of database, transaction manage, account administration, and the delegation to workgroup and meeting management functions.

7. Glossary

- **Three-tier architecture** - user interface (client), process management (server), and data management (database interface). For more details, take a look at: http://www.sei.cmu.edu/str/descriptions/threetier.html

- **CORBA**. As object-oriented software design is developing, technologies have matured and been able to support distributed object operations. This makes software development much easier and provides clear interfaces for other applications. CORBA and COM/DCOM are two widely used technologies. COM is mainly implemented on Windows, while CORBA is

implemented on various platforms and languages. They can still interact with each other. For more information, take a look at:
http://www.devdaily.com/Dir/Java/Articles_and_Tutorials/CORBA/

Obj_to_string and string_to_obj are the major functions used to stringify/destringify objects.

- **IDL**. The OMG Interface Definition Language is the language used to describe the interfaces that client objects call and object implementations provide. An interface definition written in OMG IDL completely defines the interface and fully specifies each operation's parameters. An OMG IDL interface provides the information needed to develop clients that use the interface's operations. Clients are not written in OMG IDL, which is purely a descriptive language, but in languages for which mappings from OMG IDL concepts have been defined. The mapping of an OMG IDL concept to a client language construct will depend on the facilities available in the client language.
  http://www.infosys.tuwien.ac.at/Research/Corba/OMG/idlsyn.htm#307

  JDK1.2.2 is needed for the Corba capabilities. An IDL file should be compose to indicate the interfaces that the client need to know about the server. Reference:
  http://java.sun.com/docs/books/tutorial/idl/index.html

  Corba classes implement the interfaces generated by IDL compiler: idltojava by from Sun.

- **JDBC**: for detail, look at http://java.sun.com/docs/books/tutorial/jdbc/index.html