A Web-Based System for Media Sharing and Collaborative Tools

by

Saksiri M Tanphaichitr

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000
[June 2000]

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by .....
Glorianna Davenport
Principal Research Scientist
MIT Media Laboratory, Interactive Cinema
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Web-Based System for Media Sharing and Collaborative Tools

by

Saksiri M Tanphaichitr

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

The design and implementation of *MediaExchange* examines the design and implementation of server-side software to support media sharing and collaborative tools. Enterprise Java Beans are used as a transactional, distributed, and object-relational framework for creating these tools and services. Adaptation of the server-side Model-View-Controller object-oriented paradigm for EJB is used as a basis for development. EJB architectures that maximize performance and extensibility are investigated. A complete suite of browser-based workflows is implemented, along with client libraries for accessing the system at the application logic level. Examples of application integration with the system are also given.

Thesis Supervisor: Glorianna Davenport
Title: Principal Research Scientist
MIT Media Laboratory, Interactive Cinema

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This project was designed as a part of a larger system for on-line media tools called *X-Views*, and will provide a framework for future development and applications. *MediaExchange* is the system of server-side components that provides web-based computation for these tools and applications.

MediaExchange provides a robust, concurrent, scalable system with reasonable performance that can be used as a base for development of other applications. A major objective is to provide an object-oriented interface for development of applications of similar nature.

1.1 Overview

This chapter is an introduction to MediaExchange and the motivation behind it.

Chapter two discusses technologies and choices for development and deployment of this software.

Chapter three discusses design preliminaries and analysis of design criteria.

Chapter four discusses the design and implementation of the system.

Chapter five provides examples of usage.

Chapter six identifies areas of possible improvement and further development.

1.2 Motivations for MediaExchange

MediaExchange is the successor to the server-side portion of the *I-Views* system, which formerly provided similar services to a set of browser-based clients for video manipulation, specifically, video sequencing and presentation. MediaExchange examines the requirements of I-Views and incorporates them into a more general, more robust, higher performance, and more extensible framework for its client applications and other, related applications in its class.

The server-side components have been developed alongside new client-side pieces; however, care has been taken to avoid gearing the system toward any particular application. Furthermore, the design of MediaExchange at the software level provides ease of use for future application developers who will use the platform, as well as maintainers of the code base.

1.3 Project goals and requirements

The requirements and goals of this project can be subdivided into several categories of objectives. The main goal of the project is to provide a high-level interface to a set of services. These services include user and media management, which are key underlying pieces of systems that deal with online media and media exchange. MediaExchange will provide these services while adhering to the requirements listed below.

A note should be made that security was not a primary design goal of MediaExchange. This results from the community and open nature of the system, so security in the sense of not allowing certain media clips to be shared is not implemented. All clips are visible to other users, and no clips can be removed by a user. However, access of other objects created by users can be write-restricted.

1.3.1 Concurrency and Performance

Concurrency and performance are both important to the effectiveness of an on-line transaction processing (OLTP) system. Integrity of transactions is vital when dealing with concurrent access to shared resources, but the use of overly restrictive transactions can lead to starvation between competing server processes and adversely affects performance. Managing and tuning these variables is a major and ongoing part of this system's development.

1.3.2 Scalability

In order for the system to be used well into the future, it must be easy to scale the system to match its usage. Identifying and eliminating or alleviating bottlenecks at an early stage aided in the overall design of the system, especially with regard to choices of technologies. For example, system components connecting to a shared resource simultaneously may not be a problem at smaller scales, but as the size of the system increases, this shared resource may be overloaded and become a bottleneck. Choosing which technologies to use, based on their histories in similar applications, as well as how to use them wisely, was a valuable consideration.

1.3.3 Extensibility and Maintainability

These factors are related more to the amount of engineering put into designing the code base. Major changes in underlying levels of code in the future will cause increased disruption as more systems are built upon it. Although more abstract than the other design criteria and definitely more difficult to test, adhering to good software engineering practice through close examination of software designs and dependencies before implementation will hopefully prevent major refactoring of lower-level software or any part of the existing codebase at a later date.

Furthermore, since the major goal of the system is to provide an application programming interface (API) for users, designing in an appropriate level of abstraction while allowing sufficient granularity and control is important. Extensibility, with

regard to being able to reuse the software supplied by MediaExchange, is not easy to measure, but examples of how the system can be used are supplied in Chapter 5.

Chapter 2

Technologies

The choice of technologies for implementing a system may have a direct effect on its performance, scalability, scalability, extensibility, and maintainability. Because Java has proven to be beneficial to software development productivity, particularly with regard to object-oriented design, and is mostly portable between operating sytems, it is a logical starting point for a time-constrained software engineering project. Furthermore, many server-side technologies, standards, and frameworks have recently been introduced for Java, making it a powerful and well-documented platform for developing web-based services. Although as an interpreted language it incurs a large performance disadvantage, some improvements in the system's performance can be realized through the use of run-time machine code translation (Just-In-Time compilation).

I-Views is built upon three main pieces of software (excluding the Java run-time interpreter): the Apache HTTP server; the Apache JServ Java servlet engine; and the MySQL relational database. This can be regarded as a 2-tier design, where the servlets, which accept HTTP requests and provide responses, communicate directly to the database.

MediaExchange expands upon this architecture to provide an 3-tier solution, adding a middle layer that provides data encapsulation, object-relational mapping, and workflow management, which altogether as the middle tier can be referred to as the application logic. Adding more levels of indirection and hiding lower-level func-

tionality, although in many cases adversely affecting performance, improves extensibility by removing dependencies between data and its usage. By grouping repeated and common types of access to data and encapsulating this process, applications and future application developers are shielded from lower-level details that are not pertinent to the *business logic* of the application.

2.1 Application Servers

Immediately it became clear that in order to avoid spending development time solving very difficult problems related to transactional computing, and ensuring scalability and performance, the use of a packaged application server was necessary. Not only would this provide a basis upon which to accomplish the real goals of the project—to build the system and have it in reliable working order and supporting other applications—but using a well-defined framework for the system would also make it less difficult to transfer knowledge to application developers and subsequent maintainers of the system, given that portion of the system is documented well.

Again, many solutions are available in the Java application server market; however, a small subclass of Java application servers fit the project needs closely, with regard to data handling and storage, robustness, concurrency, and scalability—those which support the Enterprise Java Beans standard.

2.2 The Enterprise Java Beans architecture

The Enterprise Java Beans (EJB) architecture was introduced recently to provide a transactional, distributed computing framework and engine for OLTP computing. An *application server* that implements the EJB standard provides a run-time system for designing and deploying objects that have several important qualities.

First, the objects are transactional. By transactional it is meant that each object can operate in the context of a transaction which can propagates to include actions on other objects, down to the database-access level.

Additionally, the objects are distributed, which means for an instance of an EJB object in the server, for cases where multiple (clustered) servers are in use, the actual object could reside on a different physical machine, and is used through network communication. Generally, all EJB communication is done over network connections, although some application servers have optimizations that revert to regular object calls when the object is local. A major use of this is for load balancing; since all objects are accessible remotely, the server can create it where resources are available.

Finally, these objects have the capability of describing an object-relational mapping. An object-relational mapping is the translation of an object's state into a database-storable form, and the process of storing it in the database. For complex and compound objects that contain information about relationships, encapsulation of this logic provides modularity such that from an object-oriented perspective there is little or no knowledge of the inner workings. These mechanisms can also be changed internally without disrupting the external use of the object.

2.3 J2EE Technologies

The EJB architecture is part of a larger initiative begun by Sun Microsystems to provide a complete server-side programming framework in Java, called Java 2 Enterprise Edition (J2EE). J2EE represents the entire family of Java technologies made for server-side programming, such as servlets, Java Server Pages (JSP), which is an extension of servlets, EJB, and underlying protocols and libraries to support these tools.

The technologies that comprise EJB are described in the following sections, along with a description of Java servlets:

2.3.1 Entity Beans

Entity beans are Java classes that are directly related to persistence (storage in a database). As described previously, some EJB objects deal with object-relational mapping. Entity beans are the objects that provide this functionality. Entity beans

provide an object-relational framework for mapping data objects to database rows. Each entity bean, in simplest form, maps directly to a database row entry, and its fields map directly to the columns of this entry. The object can be cached and managed by the application server. The server guarantees that only one instance of each entity bean (each corresponding to a row) exists and access to it is managed properly.

Entity beans can use either bean-managed persistence (BMP) or container-managed persistence (CMP). CMP means that the application server defines the database access code to load and store data and can even include generating database tables corresponding to the object fields. BMP is the lack of automatic (container- or server-managed) persistence. The entity bean itself provides all code and specific mechanisms for loading and storing data to the database, including all database queries. This not only provides more flexibility in terms of object-relational mapping, but also more performance tuning opportunities. A BMP bean is also responsible for designing its transactions, whereas CMP bean transactional code is completely generated by the server.

2.3.2 Session Beans

Session beans are EJBs not directly related to data storage and have different lifecyles than entity beans. They can be designed as either stateless or stateful objects. Stateless session beans do not store any state between uses. Therefore, they are more suited to handling utility-type workflows where no state needs to be saved. Stateful session beans are more useful for many, related requests, because they are linked directly with user sessions, and data stored in the instance of the session bean is kept as long as the user session is alive.

The application server may create many instances of a session bean and pool these instances to be able to supply many instances on demand.

2.3.3 Servlets and JSP

Java Servlets provide web-based functionality and interfaces to the rest of the system. They are used to generate HTTP-served pages and to accept dynamic requests from clients. Although not directly a part of the EJB architecture, servlets are a member of the J2EE class of technologies. Servlet engines are provided by the application server in the same runtime system and can access other server objects (e.g. EJBs) directly.

Servlets are regular Java classes, with methods that perform the service of taking in request and response objects as arguments, from which and to which the servlet can read and write data, respectively. These arguments are supplied by the servlet container that takes the HTTP request and writes it back from and to the client.

JSP is a technology for writing servlets, where dynamic content, written in Java, can be inserted into an HTML page. The servlet/JSP engine takes this template and assembles a servlet class, written in Java, compiles the class into Java bytecode, and allows it to begin handling requests. A change in the JSP code will force a code regeneration and compilation.

JSP also has unique facilities for J2EE server-side programming, such as tags to transfer control to another servlet or JSP and directives that provide automatic client session management.

2.4 Underlying technologies

2.4.1 Distributed object technologies

Much of the scalability of the EJB architecture comes from its ability to cluster server machines together as one cohesive unit. The application server software is designed so that such a cluster will act together, sharing resources and avoiding doing the same work in two places. For this reason, all EJB objects are distributed objects; that is, they may physically exist in one place, but can be accessed and used as if they were local objects from any other machine. In order to spread the workload and

resource usage, an EJB cluster will typically guarantee that only one instance of an entity bean object exists in the entire server cluster, and will pool session beans on any server. Clients will access the distributed EJB objects over the network by using Java Remote Method Invocation (RMI).

Besides the scalability benefits as described here, using RMI also allows external systems to access the server objects in the same manner. A probable use of MediaExchange in the future will be to build "thinner" systems on top of the lower-level pieces of MediaExchange that access the system objects access and store data remotely. This scenario is simplified when the protocol over which the distributed object communication takes place is standardized, and this step has already been taken in the J2EE specification.

RMI communication typically uses a proprietary protocol, Java Remote Method Protocol (JRMP), which is not compatible with other distributed object systems. Other proprietary protocols also exist; Weblogic, for their application server, uses their own version of RMI. Many newer application servers and those that conform to the J2EE standard use RMI-IIOP, which uses the Internet Inter-Orb Protocol (IIOP), of of the most commonly used protocols for distributed object communication. IIOP is the protocol for the Common Object Request Broker Architecture (CORBA), which has distributed object applications in numerous languages including C, C++, and Smalltalk. For this reason, the use of RMI over IIOP (RMI-IIOP) was preferred in designing MediaExchange, such that any application that conforms to IIOP standards can communicate with its objects.[1]

2.4.2 Naming

When objects are dispersed throughout the system and are to be used transparently, there must be a way to obtain an object reference for local use. This is done in distributed object systems through the use of naming services. Specifically, Java provides libraries for naming through the Java Naming and Directory Interface (JNDI). JNDI

[1] Although such cross-language integration through the use of CORBA is possible, doing so may require additional work.

is not limited to providing distributed object names, however; database connections and other resources can be made available through JNDI.

When the instance of a distributed object is created, in order for it to be accessible to other systems and programs it must be made available to an object request broker (ORB) and bound to a name. When a request for the object bound to a particular name is receieved, the ORB can return a reference to the object. In the context of EJB, naming is used to locate home interfaces, from which actual EJB objects can be found (using finder methods), created, or removed.

The directory part of JNDI provides organization of and contexts for naming. For all objects in an EJB server, the names of all objects in the server are bundled in one directory, the name of which is known such that a remote system can request an object from that particular directory on the server to obtain object references listed in that directory.

2.4.3 Transactional computing

Assuring that operations are *atomic*, *consistent*, *independent*, and *durable*, known as ACID transactions, is a major responsibility of an OLTP system. This occurs at the database level with database transactions provided by major database vendors such as Oracle and IBM. A major improvement over the I-Views system was to use Oracle, which is a fully transactional, row-level locking database designed for large-scale OLTP systems. I-Views' MySQL database does not support transactions.

These database-level transactions can be used by any program that connects to the database; it is the database and database drivers that provides this functionality. However, more complicated transactions may arise, for example involving two-phase commits. A two-phase commit transaction involves the use of more than one resource—e.g., two databases—where this interaction needs to be coordinated. In the event that such a transaction needs to be rolled back, all child transactions need to be rolled back as well, returning all resources to their original state. EJB systems allow such complex transactional integrity to be easily incoporated into applications, often allowing fine-grained control over creation and use of transactions within execution,

Transaction Attribute	Specifies
TX_BEAN_MANAGED	Used for bean-managed entity bean persistence or if a session bean accesses a database, where the bean handles creating/using transactions.
TX_MANDATORY	The bean requires that a transaction has already been started and will continue within this transaction. If the caller of the EJB tries to use it outside of a transactional context, an error occurs.
TX_REQUIRED	The bean can be called from within a transaction, or if not, it should start a new transaction.
TX_REQUIRES_NEW	The bean should begin a new transaction. If the caller is calling the EJB from within an existing transaction, it suspends the caller's transaction until execution returns to the caller.
TX_SUPPORTS	The bean will execute within a supplied transaction context, but can also execute outside of one, if not supplied.
TX_NOT_SUPPORTED	The bean will suspend the caller's transaction context, if one exists, and execute outside a transaction context.

Table 2.1: EJB transaction attributes.

including nesting transactions and suspending the current transaction when control is transferred to a non-transactional process.

At the EJB level, transactions can be set up in many ways. Each EJB has a deployment descriptor, which is a file describing the attributes of the EJB, including its transactional requirements. In the EJB specification, and EJB can have its transaction attribute set to one of the values listed in Table 2.1.[11] Additionally, the *isolation level* can be set for each EJB.[2] These isolation levels are applicable at the Java Database Connectivity (JDBC) library level, as they provide an interface into the transactional isolation levels of the database itself.

Three types of undesirable conditions can occur when concurrent access/update to a database is attempted. They are described in Table 2.2. The isolation levels, as they become more restrictive, prevent a larger set of errors from occuring. The valid isolation levels are listed in Table 2.3.[3][11, 6]

[2]The isolation level can actually be set separately for each method in a bean. However, this is not necessary and the bean-wide isolation level setting is used as a default.

[3]The individual isolation levels are not part of the EJB 1.1 standard, but are used in EJB 1.0. The EJB server used for MediaExchange, IBM Websphere, adheres to the latter standard.

Condition	Description
Dirty reads	A transaction reads data that has been changed but not yet committed by another transaction. The old values are *dirty*.
Nonrepeatable reads	A transaction reads a row twice; between the two reads another transaction changes the value and the second read returns a different value.
Phantom reads	A transaction reads data matching certain criteria from a table twice; between the two reads new data is added to the table that also fit those criteria. When the second read occurs, the first transaction gets the new data as well.

Table 2.2: Possible error conditions in database reads.

Isolation Level	Prevents
TRANSACTION_SERIALIZABLE	All of the above types of errors.
TRANSACTION_REPEATABLE_READ	Dirty reads and nonrepeatable reads only.
TRANSACTION_READ_COMMITTED	Dirty reads only.
TRANSACTION_READ_UNCOMMITTED	None of these types of errors.

Table 2.3: EJB transaction isolation levels.

2.5 Using EJBs

All EJB objects also have associated home and remote interfaces, in addition to the actual EJB class. The EJB class contains the business methods written by the programmer along with some generic EJB methods that must be contained in every EJB class.

When writing an object to be used as a distributed object, an instantiation of the actual class written by the user is not directly handled by the client that uses the object. Rather, when a client has a reference to a local object, it is actually handling a *stub* object that accepts the methods calls made on it, marshals the method call and its parameters into a format transferable over a network connection, and transmits this method call over a network connection to *skeleton* and *tie* objects located in the same memory space (in Java, this means the same JVM) as the actual object. These objects delegate the incoming method calls to the actual object. In the same way, they marshal return values and transmit them back to the client.

In order for the client to transparently use the stub object as if it were the real object, the stub and actual objects both conform to the same interface, which is how

the object is handled in the client—as an instance of this interface. The programmer writes this interface and the actual object class that implements this interface, and from these uses a distributed object compiler to generate the stub and tie classes.[4][10, 1]

In the context of EJB, the home and remote interfaces perform this function for two different types of distributed object classes.

2.5.1 Remote interface

The remote interface contains all the business methods of the EJB object. The client, rather than handling an instance of the actual EJB object when using an EJB, handles a stub object that implements the remote interface. All home interfaces must be subinterfaces of the EJBObject interface defined by the EJB specification.

2.5.2 Home interface

The home interface contains method signatures for a home object in the server that handles special methods for its associated EJB. It must be a subinterface of the EJBHome interface defined by the EJB specification. All home object must contain create and findByPrimaryKey methods, which both return an object of the remote interface type. The create method call creates an EJB object and returns it. Its arguments correspond to those of the ejbCreate method in the EJB object class, which initializes the EJB object. The findByPrimaryKey method takes a *primary key class* object (described below) and locates the EJB by its unique primary key. The home interface can also define other find... methods, which correspond to specific database queries to locate particular sets of entity beans conforming to the criteria

[4]The procedure described here is representative of how Java distributed objects work with CORBA. Generally, to generate CORBA objects, a programmer writes an interface using the Interface Definition Language (IDL), which can be then compiled into stub and tie classes. Since the actual object class, using this method, is not created first, some IDL compilers generate a template for the actual object class which the programmer then fills in with appropriate method bodies. Java RMI-IIOP also performs other CORBA tasks automatically, but these differences are are not discussed here.

specified by the query. The home interface also inherits a remove method from the EJBHome interface, which also takes a primary key class as an argument.

2.5.3 Primary key class

All entity beans also have an associated primary key class, which defines fields of the entity bean that together are unique over all entity beans of that type. This is closely related to the concept of database primary keys, where each database table has a field or fields that are unique over all table rows. Since each entity bean represents a database row, one or more of its fields must serve this purpose.

EJB uses this primary key object for locating (home interface find methods) and removing (remove method) entity beans.

2.5.4 Tying services together with CORBA

As well as using CORBA for distributed object services, CORBA services can be used to provide naming and transaction services as well. CORBA CosNaming and OTS, respectively, are the services provided for these uses, and using standardized architectures, again, allows external systems to not be limited to the use of Java and/or proprietary protocols.

The importance of this leads directly to the importance of choosing an application server that uses the most standardized ways of providing its services. IBM, with a strong commitment to both CORBA and Java, combines both in its Websphere application server.

2.6 IBM Websphere Application Server

IBM Websphere Application Server, version 3.02, Advanced Edition (IBM WAS/3.02 AE) was selected as the deployment system because it provides support for entity and session beans according to the EJB 1.0 standard, RMI-IIOP, Servlet 2.1/JSP 1.0 engine, and database connection management implementing Java Database Connec-

tivity (JDBC) extensions, among other facilities.

Although the J2EE platform generally uses EJB 1.1 and servlet 2.3/JSP 1.1, such packages are very new, untested, and poorly documented, whereas IBM's solution is a commercial server that has been well-tested, documented, and has a large user base and knowledge base. Also, Websphere uses a native, Apache-based HTTP server, IBM HTTP server, taking some load away from Java computation for basic file serving, and is based upon an optimized and stable Java Virtual Machine (IBM JDK 1.1.7B). The Java Virtual Machine (JVM) included in this release also includes an optimized Just-In-Time (JIT) compiler that translates Java bytecode into native machine code, allowing for performance improvement of a factor between approximately two and ten.

By supporting JDBC standard extensions, Websphere provides database connection pooling accessible through a standard Java API. IBM supplies a proprietary implementation of these classes.

Chapter 3

Design Analysis

Although EJB is a powerful framework and system for creating robust systems, it is still a young technology and common design patterns for EJB have not yet been established. Some object-oriented paradigms can be applied readily to EJB; however, the relative complexity of EJB specifications and inclusion of distributed object technologies precludes the application of many patterns. The design of MediaExchange attempts to generate and document new paradigms within the realm of EJB and distributed computing in general, particularly to minimize over-the-wire accesses with distributed objects and maximize efficiency and robustness.

The overall design of the system was well-established from previous specifications before implementation took place. However, some additional design occured to fully take advantage of the strengths of EJB and mitigate is weaknesses by applying already established paradigms for server-side software engineering as well as EJB-specific solutions. The organization of the server software, with extensions and modifications for EJB, is similar to Model-View-Controller architectures for server-side programming.

3.1 Analysis of a typical MVC architecture

A more traditional example of an MVC architecture is a graphical interface system for a computer. The parts of the Model-View-Controller were typically defined as in this example.

MVC Example: A file dialog window.

Model. The model component is the representation of data for an element. In the example, the model may represent the list of files in listed in that dialog.

View. The view component is what displays the files. In this example, it would be the construct of the window, in which the files may be presented in a list, or with icons, and there may be some additional information. The view may communicate with the model to retrieve information.

Controller. The controller component takes input requests and modifies the model and view accordingly. This may correspond to a button-handler that waits for mouse clicks on buttons in the dialog. When an event occurs, the controller makes appropriate changes to the model and view.

When moving to a web-based system, the MVC model deteriorates a bit with respect to to the traditional viewpoint. The controller and view have very different connectivity in a server-side, web-based environment. The controller cannot make immediate changes to the view, since the view is only updated on web-page refresh. This difference removes part of the functionality of the controller, since the change in view state occurs in the view. Furthermore, the controller may or may not be linked directly to control. It might receive input from the view, which will cause it to perform some action and update some aspect of the model layer.

Rather than create a circular flow of control as in the traditional MVC case, where each piece interacts with both other pieces, in the server-side case there is a hierarchical structure. The view is the only exposed part of the system to a client. However different the traditional and server-side web MVC classifications may be, the architectural model still provides useful insight into how a system should be organized.

More generally, an MVC architecure could be described as follows:

Model Layer. The model layer consists of objects that encapsulate and group data. Each model object contains methods for mutating its state. In the EJB case, a model object also encapsulates its persistence.

View Layer. The view layer consists of objects that display state in the form of web-pages. The view layer also presents the available actions for what is presented. These actions translate into actions on lower tiers.

Controller Layer. The controller layer consists of objects that encapsulate workflow and perform operations on the data through the model layer. This provides the advantage of packaging complex workflows, for example, creating an object and then associating it with other objects.

This hierarchical separation of layers provides the benefit of modularizing parts of the system. The view layer, in this case, provides the presentation of data specific to this application. The controller layer, by providing a generalized interface for manipulating data, can be exposed in APIs for programmers using MediaExchange as the lower-level basis for other systems. Together, the MVC architecture provides the application logic tier of an three-tier system.

In the EJB framework, if the controller is designed as a distributed object—specifically, an EJB object—it can be referenced remotely and used by applications housed by separate machines and systems. See Section 3.4.

3.2 Application of MVC to EJB

Established paradigms for server-side software design, especially the server-side adaptations of Model-View-Controller architecture, apply themselves differently to the EJB architecture, which has its own distinct architecture and roles for its defined objects. Although an adaptation of the MVC architecture may or may not be appropriate for use with EJB for this reason, the advantages that well-defined MVC architectures have can be leveraged in the EJB framework as well. The overall design is described here:

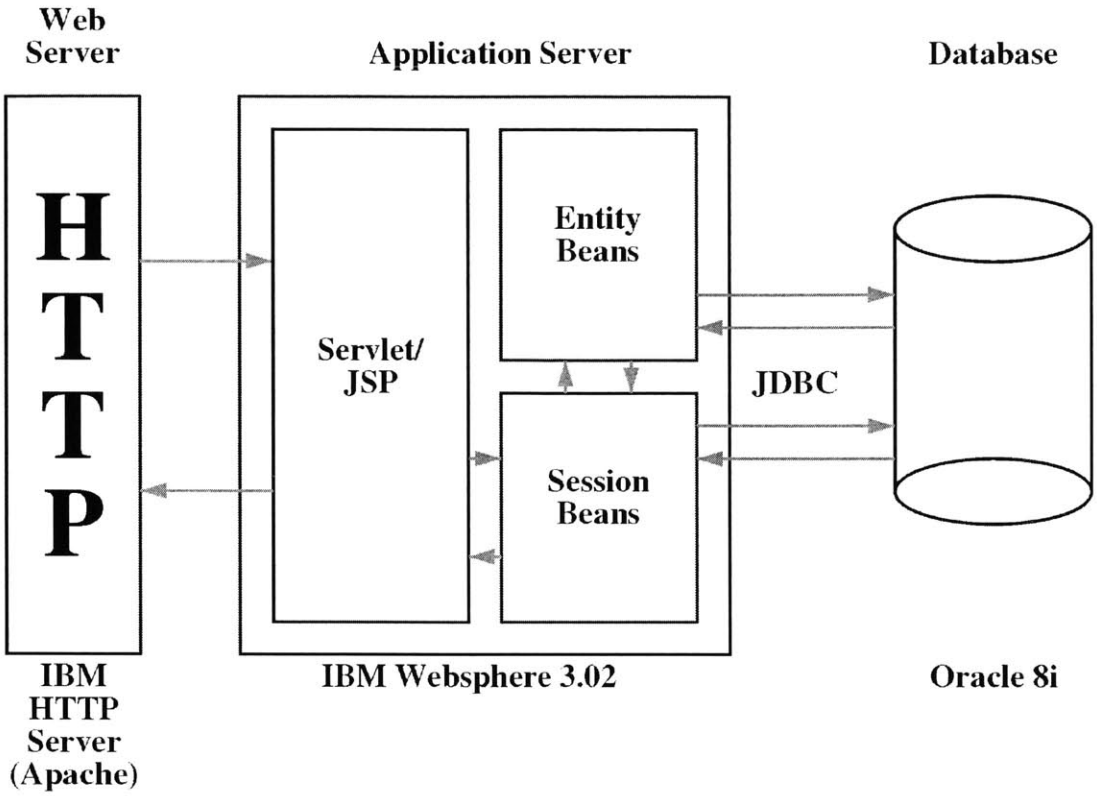

Figure 3-1: EJB architecture used in a three-tier system.

Entity beans as model objects. As discussed previously, in the EJB MVC architecture, model objects not only store data but also encapsulate the process of storing this data to a database.

Session beans as controller objects. With the definition of controllers as bundles of workflow, using a session bean as a stateless interface to workflow comes with many advantages. Again, complex workflows that appear often can be modularized, and the interface can be exposed as not only an API, but also to remote systems and applications.

Servlets and JSP as view objects. Servlets (and JSP pages, which compile into servlets) are used as the interface to web browsers and web-based clients to generate content and accept browser-type requests and workflows.

3.3 Common EJB designs applied

Beyond the application of MVC to the EJB server-side software as a general methodology, improvements can be made as to how EJB objects are handled. These improvements provide better object-oriented design, and may improve performance.

3.3.1 Session bean as a facade to entity beans

The entity beans themselves should not be handled directly by clients (here referring to any system that has code-level access to the EJBs). Rather, a session bean that defines the workflow appropriate for a particular entity bean or set of entity beans should be used, which will in turn handle the entity bean finding, creation, removal, and business methods. Although this level of indirection may, again, decrease performance, it allows better modularity and simplified application programming, especially by those who are not familiar with the underlying workings of the system. The session bean interface to the lower levels of the system provides a simpler way to access entity beans, for example, supplying an object primary key by which to locate the object for a desired operation, rather than going through the process of finding the Home object for the entity bean, calling the appropriate finder method, and handling lower-level exceptions that may be raised for errors. The session bean can handle this process, and translate the lower-level exceptions into ones more meaningful at the application level. This type of structural pattern is called a *facade*.[5]

Also, the session beans can handle more complex workflows. A relationship between two entity beans can be made transparent to the application developer by supplying a session bean that acts as a facade to both sets of beans, performing the desired operations without requiring the application programmer to know explicitly what is occuring with each entity bean. This type of indirection is also important given that the relationships or the entity beans themselves may change. Encapsulating and hiding the inner workings of this workflow is important to assure minimal or no changes to application-level code if change need to be made at workflow or lower levels.

This specification is consistent with the MVC methodology previously described. Also, when the session bean is exposed as an API, it also provides all possible workflows that can be performed on entity beans such that the client program does not access the entity beans directly and provides a higher level of abstraction.

3.3.2 Business Interfaces

Although the actual implementation class of the EJB class is required by specification to implement all business methods defined in the EJB remote interface, this requirement is not enforced at source-code compile time, as there is no direct object relationship between the implementation and interface. This can be solved by using a *business interface*, which the remote interface will extend and the implementation class will implement.[4] This requires that the implementation class at least implement the methods contained in the business interface, and provides an assurance at compile-time that the requirements are met for the subsequent compilation into the objects actually used at run-time by the EJB server.

Business interfaces are also allow for increased flexibility in terms of what objects can be used by higher levels of the application. For example, when the web layer of the application accesses a user, it performs a method call on a session bean and asks for an object that is only required to conform to the business interface. This object could either be the actual EJB or a state object. The state object will be returned to the view/client layer, and if any changes need to be made to the actual object, the client can use a session bean method that modifies the entity bean.

A more advanced use of the business interface may be to use a proxy object that conforms to the business interface specifications and acts primarily as a holder object, but can delegate changes down to the lower layers by accessing an appropriate session bean. This approach, however, is beyond the scope of this thesis and is not explored further.

3.3.3 State objects

A state object is used as a holder object for data. The state object can be used as a cached data set that holds the record locally, without incurring the cost of many remote method calls (one large access is better than many smaller accesses) or the overhead from dealing with an entity bean.

3.4 Access to MediaExchange

Since the main purpose of MediaExchange is to provide a universal repository and management system for media clips and their users, flexibility in how a client application can connect to the server is essential. The methods by which a client interacts with MediaExchange are described here.

3.4.1 CORBA/RMI-IIOP access to session beans

Since session beans (and all EJB objects) use the CORBA standard for distributed object communication, a natural way to access the functionality encapsulated in session beans is to access them directly as distributed objects. This method has no regard for whether the client application is hosted on the same server or on a physically separate machine, as long as the two applications share a network connection.

3.4.2 HTTP-based communication

Servlets and JSPs can be implemented to communicate with HTTP-based clients through the web. This allows clients to be completely language-independent as long as they can open and transfer data over HTTP connections to the web server. A set of Servlets/JSPs can be implemented to perform specific queries and responses to and from the server. This results in text-based communicaton over HTTP. The text can be structured in numerous ways, whether over proprietary structures designed per application or over more general structures such as the extensible markup language (XML). The advantage of using XML is that XML parsers are available for most

languages and any type of hierarchical data can be structured in XML. However, this incurs the disadvantage of requiring a client to have an XML parser available, and for applets, which download all data from the server at run-time, this can be expensive. For this reason, it may be better to depend on Java libraries already available to the client and use proprietary forms of text messages that can be decoded by such libraries.

Chapter 4

Design of MediaExchange

This chapter provides an overview of the design of MediaExchange. Critical EJB functionality is described in detail. Exposed controller interfaces (session bean interfaces) are supplied in Appendix A.

4.1 Database primary key/ object identifier uniqueness

Database primary keys, or in object-oriented terms, object identifiers (OIDs) are important to any relational or object-relational system. Object IDs need to be separate from business logic and unique across the system. That is, the primary key for any data object will be an OID, and never any piece of information that is specific to the data. This allows the software to remain independent of any application-specific data.[2]

OIDs should also be unique across the entire system. If two tables were to keyed by the same set of OIDs, and these keys were used as values in a separate table in the same column, conflict could occur.

Generating a unique key can be accomplished in several ways. One solution would be to use an auto-numbering scheme supplied by the database, which would generate IDs that are unique within a database table; however, this does not meet the

requirement that object IDs are unique across the entire system. A more appropriate solution is to use a single object instance (a Singleton object)[5] that remains loaded and singular for as long as the JVM is running.

This Singleton object could generate unique OIDs by storing in the database the last generated OID, and reading this value and incrementing it per request. However, this approach leads to a database query (or two, depending if there are two requests—one to read the old value; one to write the updated value) for each OID request and could lead to a variety of transactional problems when a large number of resources request new OIDs. Completely serializing the OID generation using sychronization could lead to starvation between threads. A better approach would be to only look periodically in the database, while still guaranteeing OID uniqueness, and this is accomplished by splitting the OID into high and low segments, and storing the upper segment in the database, keeping the low segment in memory and incrementing it in memory. When the low OID reaches its maximum value, the upper OID value is incremented. This minimizes the computation necessary to generate each OID, allowing the process to be sychronized at some level. To guarantee uniqueness, each time the server is started the high OID is incremented, such that no matter what low OID it is paired with, that resulting OID has never been used. When the server is started, the low OID starts at zero. This results in many OIDs being "wasted," but given that the high OID segment is large enough, the system will never deplete its OID supply. In MediaExchange, all OIDs are 64-bit numbers, split into two 32-bit Java ints. The maximum value of an int is $2^{31} - 1$, over two billion[1]. To extend the number of server restarts possible, the high portion of the OID could be extended into multiple 32-bit fields, resulting in a 96- or ever 128-bit OID. However, to maximize performance, in this scheme the OID is kept short. Even still, with over two billion server restarts and over 4×10^{18} OIDs available, a 64-bit OID is more than sufficient. The ints are converted to Strings, simplifying the concatentation of the high and low portions. Also, to shorten the length of each OID string, each sub-string is the hexadecimal representation of the int.

[1]The maximum value of a signed two's complement 32-bit integer.

This is appropriate for single-virtual-machine environments, but an application server cluster supports at least one VM per machine. This requires that the single object instance is reachable and singular across all machines in the cluster. The solution for this is to create a distributable object that runs on one machine. Any machine in the cluster can generate a reference to this object and obtain the next unique OID.

This OID generator is bound in the same naming context as EJBs to simplify locating it.

4.2 User management

The user management portion of the system defines classes that describe users and workflows for user creation, validation (upon login), and deletion. A user object has the following fields:

UserEJB **entity bean.** This is a container-managed entity bean that contains oid, username, password, last name, first name, email, country, and age fields. Its home interface defines the additional finder method findByUsername which locates all User entity beans with a given username. However, since no registration of users with duplicate usernames is allowed, this method should always return one user. It is used for logging users in (the user object is found by username, and the supplied password is validated against the password stored by the object).

Initially, the password was to be stored in the database as a 128-bit MD5 one-way hash of the password string for additional security, but incompatibilities with Websphere's CMP handling of byte arrays (the Java byte[ ] array type) and the database storage of byte arrays/octet streams (Oracle RAW data type) precluded using this solution.

Figure 4-1 shows a UML diagram of the business interface hierarchy for the User business interface and associated objects.

UserController **session bean.** This session bean contains methods for creating users, validating users, and other operations that can be performed on entity beans.

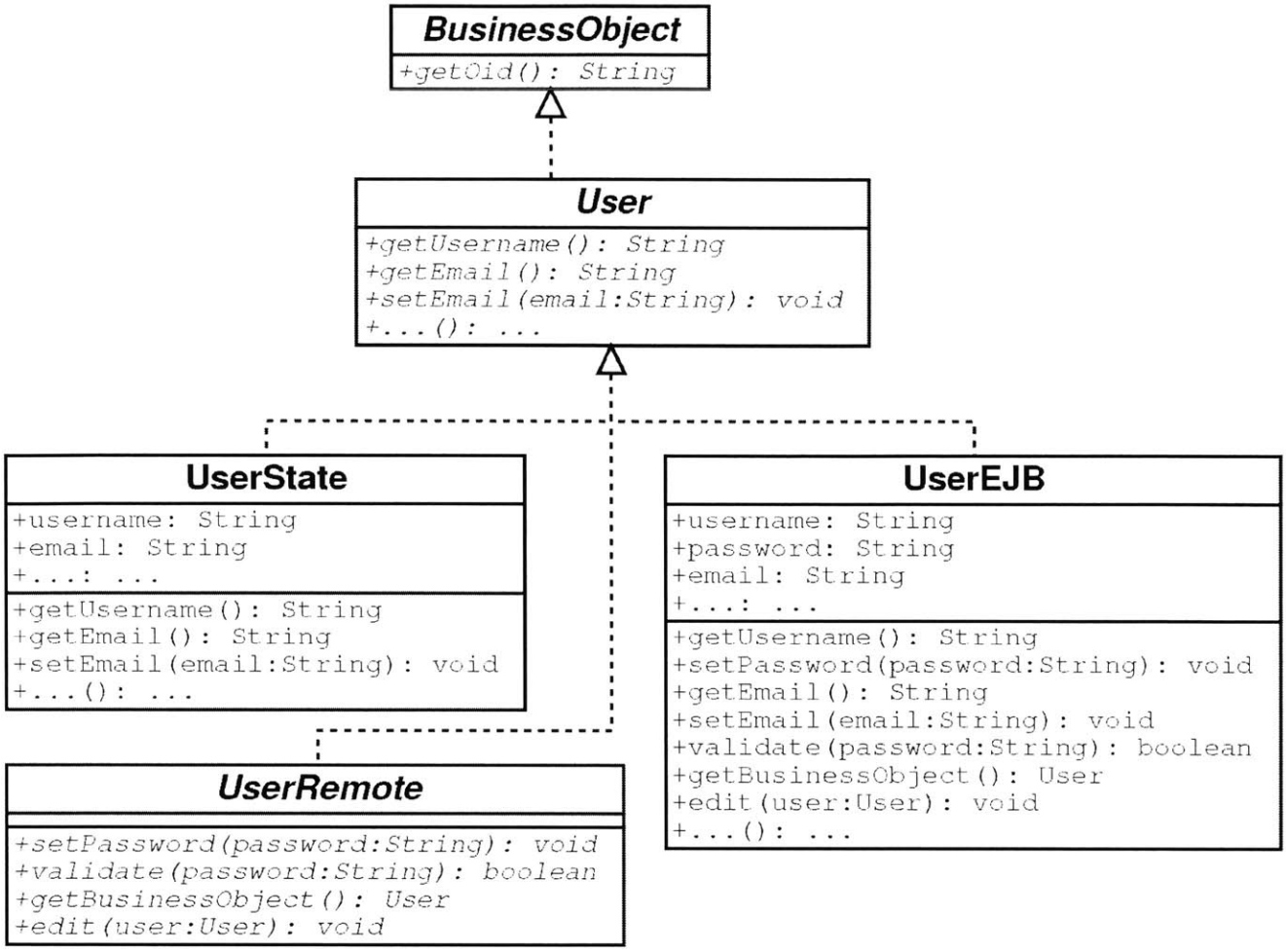

Figure 4-1: User Business interface/EJB/State object UML diagram.

Refer to Appendix A for details.

UserState **state object.** The UserState object is a holder class that contains all the user data stored by the UserEJB object. Rather than use the UserEJB object, an application programmer can retrieve the UserState object instead and avoid using a network-intensive distributed object to access data.

Note, however, that the state object contains no password field. This was done for security reasons. The actual entity bean must be accessed to change the password or validate the user; these methods are not contained in the User business interface but rather in the EJB remote interface UserRemote.

User **business interface.** As described in the design analysis, both the state object and the entity bean for a model object are inherited from a common business

interface. The User business interface defines methods that are common to both.

UserRemote **entity bean interface.** This extends the User interface to provide all the methods of the actual entity bean. It provides the business methods not contained in User, as well as a getBusinessObject method to obtain a UserState object.

4.3 Media clip management

Media management is handled by a similar object hierarchy and tier structure. All different types of media will be handled by a single type of media clip object.

MediaClipEJB **entity bean.** This is also a CMP entity bean. It contains fields for the owner OID, the title, description, type, URL, fileSize, and a boolean flag that is used to mark whether a clip is stored locally.

MediaClipController **session bean.** Similar to UserController, this EJB provides business logic. Refer to Appendix A for details.

MediaClip **business interface.** This interface provides all media clip business methods, except for getBusinessObject which returns a MediaClipState object.

MediaClipState **state object.** This class stores metadata for a media clip.

4.4 Object tags

Object tags are used as a way to annotate objects with information. A common tag attached to objects may group objects together, or contain specific information about an object. The use of such annotation is directed mainly toward organizing MediaClip objects, where they hold information not contained in the object itself.

Tags are implemented as a simple relational mapping scheme, where a main tag table stores all tags for all media clips. Each tag data entry references a media clip OID, a tag type, the OID of the user who added this tag, if any, and the tag data. The design of the sytem is such that no data is hidden from simple table queries. An example of hiding such data is in the case of a one-to-many relationship between an entity bean other objects. If the entity bean needs to store a list of related OIDs,

ClipOID	UserOid	TagType	TagBody

Table 4.1: Tag entry database structure.

for example, it could create a Vector object, insert all the values of the list, serialize this data into a byte-stream, and store in in the data in this binary form in the database. However, this relationship is not searchable because the data needs to be decoded back into its Java object form, and for this reason all relationships are stored in searchable mapping tables that pair objects that have a relationship in table rows.

The motivation behind this structure is to allow extensibility in how information about a MediaClip is represented. If a new type of information needs to be added to a MediaClip, the structure of this information is similar to any other information that MediaClips are "tagged" with. Thus, the flexible and simple scheme of relating a clip to possibly many tags is useful and extensible. Currently, the media clips are tagged with keywords, comments, and channels.

Furthermore, because the tags are closely integrated with the database, advanced queries can be performed across tables using appropriate SQL queries. This also has the advantage of improving performance, for in a one-to-many relationship, to explore such a relationship in the EJB context might mean mapping a Java object for each child object, when this may not be necessary. This concept of "lazy evaluation" of object with regard to loading entity beans will improve performance by decreasing object creation overhead for simple data accesses.

However, the interface to the tag data is not directly through SQL queries. Rather, stateless session beans are provided for this purpose, performing the query and returning data sets. For searching tags, session beans are used to make queries and return data without incurring the cost of instantiating entity beans. When the entity bean object is needed (for modifying bean data, for example), the application can "drill down" and then find the EJB object for use.

4.4.1 Keywords and comments

Keywords are implemented as object tags, where any number of keywords can be associated with a particular object. Comments are also implemented as object tags.

When a clip receives a keyword or comment, an entry in the tag table will be created, mapping the keyword or comment string with the OID of the clip. Then, a database query can be used to retrieve all the keyword and comment data for a clip.

A session bean, TagController, is used to access this specific functionality for keywords and comments, as well as general tag addition and deletion. Refer to Appendix A for details.

4.5 Channels

Although channels are similar to keywords and comments in that a tag can be used to group clips together, channels have more structure and information. For example, while any registered user can add keywords and comments, a channel has an owner, a flag to determine if it is public or private (if other users can add clips to the channel), and other metadata that would not be easily stored only by a set of tags.

Channels are represented by both entity beans and object tags. The entity bean (CMP) stores the object metadata, and a tag query is used to retrieve metadata of the clips that are tagged with this channel's OID. When a channel is deleted, these associations are removed from the tag table as well.

Specifically, the tag type for channels is "CHANNEL", and the user OID specified by the tag is that of the user who added the clip to the channel.

Channels are also used to implement the concept of a home channel, or where each user stores his or her own clips. Upon user registration, a home channel is created for the user and the user has access to this channel upon login. Clips can be added and removed from this channel, and other users can search all other users' home channels. Although this open specification carries the implication that nothing is truly private in the system, extensive security was not a primary goal of the system, given that MediaExchange and associated applications are designed as community-

based systems where users share all their data.

4.6 Search

The search facilities of the system will examine any data held in the system database tables. Currently, the system provides facilities for searching media clips and their tags (keywords, channels, comments), if any. Users are comprised of only the data in their corresponding entity beans; only a findByUsername is supplied in the EJB home interface. No tag functionality has been implemented for users.

MediaClips can be searched in several ways. First of all, they can be searched based on basic metadata, such as title, description, and creator. Second, the comments or keywords on a media clip can be searched by any such tags attached to the media clips. Finally, because clips can be organized into channels, it can be determined which clips are in a channel.

MediaExchange currently only supplies search capabilities at the application level; no views for the web interface have been implemented.

4.7 Web interface design

The web interface is mainly for user registration, adding media to the system, and user media management (adding/removing clips from a channel or creating a channel, for example). Most applications will reach the system at the tier below the web interface, accessing the session beans directly.

The only exceptions are applications that require an HTTP interface, notably applets, where distributed object communication is possible, but large network lag times may dictate a stateless type of communication where each data request is separate and requires shorter, more specific connections to the server. In such cases, Servlets and JSPs can be used to transfer text-based messages to and from the server as described in section 3.4.2.

4.7.1 Login and portal pages

The initial portal page for the MediaExchange web interface is viewable by any user. However, the priveleged and customized interface for registered users is unavailable until login has been completed. The header bar of the portal page displays either a login form (username and password boxes) or a logout button. When a priveleged operation is accessed without login, the user is directed to a login page.

The first time a user opens a page at MediaExchange, a session is created and a cookie, a name/value pair, is given to the client, which stores it locally. On every subsequent request, the client returns the cookie, which contains a session ID. This session ID maps to the session created previously. Upon successful login, a User object is stored with the session on the server, and when a priveleged view is accessed, the page ensures that the User object is present in the session.[2]

Sessions are set to time out after 1800 seconds, after which all objects stored with the session are invalidated and cleaned up. The user must log in again after this length of inactivity.

4.7.2 User registration

User registration pages accept desired username and password information as well as other user data. Upon successful registration (all information supplied, unique username chosen), the user is stored in the database and a home channel is created for the user.

4.7.3 Uploading media

Media can be added to the system at will by any user. The only requirement is that the media clip is smaller than a maximum size of ten megabytes (arbitrarily set and changeable), and that the user is registered and logged into the system.

[2]If the client does not support cookies, the server can incorporate a session ID into every link the server writes to the client in each page. This method of session tracking, called *URL rewriting*, can be supported by all browsers. However, at this time clients must support cookies.

The upload workflow begins with a form for the media clip title and description. Creating an object for this new clip, it then proceeds to the actual upload page, in which the user selects the local location of the contents of file. Once submitted, the browser sends a MIME multipart/mixed message containing the file and its meta-information, including size and type. Once the clip is uploaded, the type, size, and URL fields are added to the clip EJB. The URL is dependent on the type of clip. RealMedia clips (`*.ram, *.rm`) are streamed by a Real streaming server and have URLs prefixed with `pnm://` and and are located at a different path. Quicktime (`*.mov`) and MPEG videos are streamed by the HTTP server. All clip are stored and referenced by their OID, plus a file extension that allows clients to differentiate between media types and open an appropriate browser.

Finally, the clip is added to the user's home channel. Users can also choose at this point to add thumbnails, keywords, or comments to the clip.

4.8 Client libraries

Client libraries have been developed for connecting to the server at either the web tier or the application tier (at the session bean level). They are described in detail here, and examples are given in Chapter 5.

4.9 Web-based connection library

Java library classes provides a connection to the server for applets. The library is initialized with several applet parameters embedded in the applet HTML page, including the server path and the user OID. The user's home clips are by default also embedded as a colon-delimited list in an applet parameter. More sophisticated client-based searches have not been implemented and there are no client library functions to support them.

Since applets are not expected to support cookies, the web connection library does not include session tracking functionality. Generally, however, the client application

only needs to download clip metadata, which can be done without logging in. In the case that the client application needs to upload data, such as a SMIL file constructed on the client, it suffices to submit the user's OID with the request. Again, the goal of MediaExchange is not to provide high levels of security. User tracking is mainly for community/collaboration information.

The library currently provides clip metadata downloading and SMIL file uploading capabilities.

Metadata downloading is provided by the use of a client-side Clip object, which, upon instantiation with an OID, makes a server request to a dedicated JSP for the metadata for the clip with this OID. Following successful instantiation, the Clip object's fields are populated with the metadata and the client makes local method calls to get the field data.

For uploading clips, the library opens a connection to the server for a different JSP using a Java library, uploads the SMIL string, the user OID, and title. In response, the client receives a URL which can then be used for previewing/viewing the uploaded file. On the server side, the receiving JSP writes the SMIL string to a file, such that it can be separately served by the HTTP server.

4.10 Session bean connection library

The MXControllerUtil Java library provides a connection to session beans. A properties file packaged with the library must be located somewhere on the system, and this path must be supplied to the library upon initialization.

This utility provides a method for initializing with a properties file, and locating JNDI-named objects at a server specified by the properties file. The method can be used to locate an EJBHome object for a specific session bean, which can then be used to create the appropriate controller (using the create method). Using the exposed session bean APIs, the client can then perform method calls to operate on data at this level.

Chapter 5

Using MediaExchange Libraries

This section gives insight into how well MediaExchange performs the tasks of providing its services to client applications, in terms of ease of use.

For usability testing, a case study of an X-Views applet that transfers data over HTTP is included. Also, a hypothetical example of an application that communicates with MediaExchange at the session bean level over RMI-IIOP/CORBA using Java libraries. No applications utilize this functionality at the moment, but some are planned for the near future.

Libraries are specified in more detail in Appendix B.

5.1 Web-based library connection example

The use of the web-based library is straightforward. The supplied Clip object handles the population of media clip metadata fields for each object by performing a server request.

This example takes a string consisting of a colon-delimited list of OIDs from the applet parameters, creates a Clip object for each OID, and stores each resulting Clip object in a Vector object for later use.

It also uploads a SMIL string to create a clip on the server. It then updates this clip with another SMIL string.

```
import edu.mit.media.ic.lib.MXWebUtil;
import edu.mit.media.ic.lib.Clip;
import java.util.*;

public class ... extends Applet {

    Vector clips;

    public void init() {
        clips = new Vector();
        StringTokenizer st = new StringTokenizer(getParameter("sequence"),
                                                 ":", false);
        while (st.hasMoreTokens()) {
            /* create a new clip for the next OID in list,
             * add it to Vector of clips
             */
            clips.addElement(new Clip(st.nextToken()));
        }


        String smilString = "...";
        String server = getParameter("SERVER_URL");
        MXWebUtil.init(server);
        String smilOid = null;
        try {
            smilOid = MXWebUtil.uploadSMILFile("my clip", smilString);
            Clip smilClip = new Clip(smilOid);
            MXWebUtil.updateSMILFile(smilOid, smilString);
        } catch (MXWebException e) {
            e.printStackTrace();
        }
    }
}
```

5.2 Session bean connection example

The properties file used with the library can be edited by the user to specify the location of the server. It also contains entries for the JNDI names that it can locate. The library must be initialized with local location of the properties file, after which it can be used to locate controller objects.

The following is a code example of how to locate and use a remote controller. Refer to Appendix A for specifications of available controllers.

```
import edu.mit.media.ic.lib.MXControllerUtil;
```

```
public class ... {

...

  // initialize library with properties file location
  try {
    MXControllerUtil.init("d:\\MediaExchange\\mx.properties");

    /* get UserController object by getting home and performing
     * create method on home
     * must cast to home type and specify <HomeClassName>.class
     * as second argument to getHome method
     */
    UserControllerHome ucHome =
      (UserControllerHome) MXControllerUtil.getHome("UserControllerHome",
                                                    UserControllerHome.class);
    UserController uc = ucHome.create();

    // log in a user using the userValidate(username, password) method
    User user = uc.userValidate("a_user", "the_password");

    // get a MediaClipController
    MediaClipControllerHome mcHome =
      (MediaClipControllerHome)
        MXControllerUtil.getHome("MediaClipControllerHome",
                                 MediaClipControllerHome.class);
    MediaClipController mc = mcHome.create();

    // find a clip with oid 00000004000000A0
    String oid = "00000004000000A0";
    MediaClip clip = mc.findMediaClipByOid(oid);

    /* remove this clip, will throw an AccessFailedException if user is
     * not owner
     * User calling method is supplied as first argument
     */
    mc.mediaClipRemove(user, oid);

  } catch (CreateException e) {
    // catch creation exceptions from session bean creation
    e.printStackTrace();
  } catch (RemoteException e) {
    // catch all remotely thrown exceptions that might arise
    // from controller operations
    e.printStackTrace();
  }

...

}
```

Chapter 6

Conclusion

MediaExchange provides a rich set of tools for the media sharing application programmer. The supplied APIs, particularly the TagController interface, provide a high level of flexibility for designing applications while not concerning the user with too much low-level detail.

Server testing should be performed on a scaled-up deployment of MediaExchange. The development machine was computationally overloaded and probably does not provide performance representative of the design's capabilities. See C for development machine details.

6.1 Future implementation

This thesis provides a base for implementing a set of user and media clip management workflows. It provides basic functionality (user, clip creation, association), but more advanced and user-friendly functionality should be implemented, such as user profile edit consoles and better clip searching tools. The latter can be created using the object tag mechanism described in Section 4.4, and appropriate JSP and session beans should be implemented to support such logic. Since the goal of this thesis was to develop the underlying systems and libraries for accessing them, only a minimal level of web-based access for user validation and media management was implemented.

The idea of object tags was only explored for use in keywords and comments,

but the use of general annotation of objects in the system could be more extensive. Rather than force application developers to create new tables and data models to add annotative information, object tags will allow developers to use the existing object-oriented code base for future development. Furthermore, when new object tags are created to construct a new association between media and/or users, new session beans can be created to expose an API to this functionality at the object, and not data level. Current session beans can also be improved and geared toward specific applications by adding new or changing existing workflows defined by session bean methods.

Integration of CORBA objects with other languages could also be explored. While the system does provide a high level of flexibility for application logic and design, it could also do so for a range of languages. This would involve creating general IDL interfaces for the objects and method argument and return types from the existing Java code, and compiling the IDL into stubs and skeleton classes for other languages.

Overall, the direction of future implementation should be to expand the application codebase and web interface.

Appendix A

Object Specifications

This appendix contains method specifications for all important classes, which include the user and media clip management classes. Some specifications are included for utility classes.

`import` statements for non-MediaExchange classes are omitted.

A.1 User classes

A.1.1 User business interface

```
package edu.mit.media.ic.mx.principal.business;

import edu.mit.media.ic.mx.principal.Principal;

public interface User extends Principal {

    /* provides all getter/setter methods for
     * business object fields (all except password)
     */
    public String getUsername() throws RemoteException;
    public String getEmail() throws RemoteException;
    public String getNameLast() throws RemoteException;
    public String getNameFirst() throws RemoteException;
    public String getGender() throws RemoteException;
    public String getCountry() throws RemoteException;
    public int getAge() throws RemoteException;
    public void setEmail(String email) throws RemoteException;
    public void setNameLast(String nameLast) throws RemoteException;
    public void setNameFirst(String nameFirst) throws RemoteException;
    public void setGender(String gender) throws RemoteException;
    public void setCountry(String country) throws RemoteException;
    public void setAge(int age) throws RemoteException;

}
```

Note: Principal is an empty interface that extends BusinessObject:

```
package edu.mit.media.ic.mx.businessobject;

public interface BusinessObject {

    public String getOid() throws RemoteException;

}
```

A.1.2 UserController session bean

```
package edu.mit.media.ic.mx.principal.controller;

import edu.mit.media.ic.mx.principal.business.User;

public interface UserController extends EJBObject {
    /* Creates a user specified arguments,
     * returns a User business object.
     */
    public User userCreate(String username, String password,
                           String email, String nameLast,
                           String nameFirst, String gender,
                           String country, int age)
        throws RemoteException, CreateException;

    /* Removes user with OID <oid> from database.
     */
    public void userRemove(String oid) throws RemoveException, RemoteException;

    /* Finds user <username>, checks that given password matches,
     * and returns User business object if successful.
     * throws AccessFailedException if passwords do not match.
     * throws PrincipalNotFoundException if no user <username> is found.
     */
    public User userValidate(String username, String password)
        throws AccessFailedException, RemoteException,
               PrincipalNotFoundException;

    /* Accepts User business object <user> as argument,
     * updates user metadata with metadata stored in <user>, except for password,
     * which is not stored in the User business object.
     * throws PrincipalNotFoundException if no user <user>.getOid() is found.
     */
    public void userEdit(User user)
        throws RemoteException, PrincipalNotFoundException;

    /* Finds the user object for user with OID <oid> and returns
     * User business object if successful.
     * throws PrincipalNotFoundException if no user with OID <oid> found.
     */
    public User findUserByOid(String oid)
        throws PrincipalNotFoundException, RemoteException;

    /* Finds the user object for user <username> and returns
     * User business object if successful.
     * throws PrincipalNotFoundException if no user with OID <oid> found.
     */
    public User findUserByUsername(String username)
        throws RemoteException, PrincipalNotFoundException;

}
```

A.2 Media clip classes

A.2.1 MediaClip business interface

```
package edu.mit.media.ic.mx.media.business;

import edu.mit.media.ic.mx.businessobject.BusinessObject;

public interface MediaClip extends BusinessObject {

    public String getTitle() throws RemoteException;
    public String getDescription() throws RemoteException;
    public String getType() throws RemoteException;
    public String getOwner() throws RemoteException;
    public String getUrl() throws RemoteException;
    public int getFileSize() throws RemoteException;
    public boolean isLocal() throws RemoteException;

    public void setTitle(String title) throws RemoteException;
    public void setDescription(String description) throws RemoteException;
    public void setOwner(String owner) throws RemoteException;
    public void setUrl(String url) throws RemoteException;
    public void setFileSize(int size) throws RemoteException;
}
```

A.2.2 MediaClipTag state class

MediaClipTag extends MediaClipState, which implements the MediaClip business interface. That is, MediaClipTag also conforms to the MediaClip business interface. It provides the following additional methods (method bodies, instance variables, constructors omitted).

```
package edu.mit.media.ic.mx.media.business;

public class MediaClipTag extends MediaClipState {
    ...
    public String getTagType() throws RemoteException;
    public String getTagBody() throws RemoteException;
    public String getTagOwner() throws RemoteException;
}
```

A.2.3 MediaClipController session bean

```
package edu.mit.media.ic.mx.media.controller;

import edu.mit.media.ic.mx.media.business.MediaClip;
import edu.mit.media.ic.mx.principal.business.User;
import edu.mit.media.ic.mx.util.*;

public interface MediaClipController extends EJBObject {

    /* creates a media clip with specified arguments.
     * only title has to be non-null.
     */
    public MediaClip mediaClipCreate(String title, String description,
                                     String type, User caller, String url,
                                     int fileSize, boolean local)
        throws RemoteException;

    /* removes media clip with OID <oid>
     * should only be used for removing temporary SMIL files.
     * throws ObjectNotFoundException if media clip not found.
     * throws AccessFailedException if owner OID of clip not <caller>.getOid()
     */
    public void mediaClipRemove(User caller, String oid)
        throws AccessFailedException, ObjectNotFoundException, RemoteException;

    /* sets title for media clip with OID <oid> to <title>
     * throws ObjectNotFoundException if media clip not found.
     * throws AccessFailedException if owner OID of clip not <caller>.getOid()
     */
    public void mediaClipSetTitle(User caller, String oid, String title)
        throws AccessFailedException, ObjectNotFoundException, RemoteException;

    /* sets description for media clip with OID <oid> to <description>
     * throws ObjectNotFoundException if media clip not found.
     * throws AccessFailedException if owner OID of clip not <caller>.getOid()
     */
    public void mediaClipSetDescription(User caller, String oid,
                                        String description)
        throws AccessFailedException, ObjectNotFoundException, RemoteException;
```

```
    /* sets new owner OID for media clip with OID <oid> to <ownerOid>
     * throws ObjectNotFoundException if media clip not found.
     * throws AccessFailedException if orignial
     * owner OID of clip not <caller>.getOid()
     */
    public void mediaClipSetOwner(User caller, String oid, String ownerOid)
        throws AccessFailedException, ObjectNotFoundException, RemoteException;

    /* sets URL for media clip with OID <oid> to <url>
     * throws ObjectNotFoundException if media clip not found.
     * throws AccessFailedException if owner OID of clip not <caller>.getOid()
     */
    public void mediaClipSetUrl(User caller, String oid, String url)
        throws ObjectNotFoundException, RemoteException;

    /* sets file size for media clip with OID <oid> to <size>
     * throws ObjectNotFoundException if media clip not found.
     */
    public void mediaClipSetFileSize(String oid, int size)
        throws ObjectNotFoundException, RemoteException;

    /* finds media clip with OID <oid>
     * throws ObjectNotFoundException if media clip not found.
     * throws AccessFailedException if orignial
     * owner OID of clip not <caller>.getOid()
     */
    public MediaClip findMediaClipByOid(String oid)
        throws ObjectNotFoundException, RemoteException;

    /* finds media clips with titles like <title>
     * returns empty array (length=0) if none found
     */
    public MediaClip[] findMediaClipByTitle(String title)
        throws RemoteException;

    /* finds media clips with descriptions like <description>
     * returns empty array (length=0) if none found
     */
    public MediaClip[] findMediaClipByDescription(String description)
        throws RemoteException;

    /* finds media clips of which ownerOid is <ownerOid>
     * returns empty array (length=0) if none found
     */
    public MediaClip[] findMediaClipByOwner(String ownerOid)
        throws RemoteException;
}
```

A.3 TagController session bean

```
package edu.mit.media.ic.mx.media.controller;

import java.rmi.RemoteException;
import javax.ejb.*;
import edu.mit.media.ic.mx.media.business.MediaClipTag;

public interface TagController extends EJBObject {

    /* finds all clips with keyword like <keyword>
     * returns zero-length array if none found
     */
    public MediaClipTag[] findClipsByKeyword(String keyword)
        throws RemoteException;

    /* finds all clips with comment like <comment>
     * returns zero-length array if none found
     */
    public MediaClipTag[] findClipsByComment(String comment)
        throws RemoteException;

    /* finds all clips with any of tag types contained in <tagTypes>
     * with tagBody = <tagBody> if <exact>
     * or tagBody like <tagBody> if not <exact>
     * returns zero-length array if none found
     */
    public MediaClipTag[] findClipsByTag(String[] tagTypes, String tagBody,
                                         boolean exact)
        throws RemoteException;

    /* finds all clips by with ownerOid <ownerOid>
     * returns zero-length array if none found
     */
    public MediaClipTag[] findClipsByTagOwner(String ownerOid)
        throws RemoteException;

    /* adds tag with specified arguments
     * if not <allowDup>, does not allow duplicate tags
     * does nothing if tag already exists
     */
    public void addTag(String clipOid, String ownerOid, String tagType,
                       String tagBody, boolean allowDup)
        throws RemoteException;
```

```
    /* adds keyword <keyword> to clip with OID <clipOid>
     * with owner <ownerOid>
     */
    public void addKeyword(String clipOid, String ownerOid, String keyword)
        throws RemoteException;

    /* adds comment <comment> to clip with OID <clipOid>
     * with owner <ownerOid>
     */
    public void addComment(String clipOid, String ownerOid, String comment)
        throws RemoteException;

    /* removes all tags for clip with OID <clipOid>
     */
    public void removeAllTagsForClip(String clipOid)
        throws RemoteException;

    /* removes tags matching tagType <tagType> and
     * exact tagBody <tagBody> for clip with OID <clipOid>
     * should only be used for channels and other object-to-object mappings
     */
    public void removeTagsForClip(String clipOid, String tagType, String tagBody)
        throws RemoteException;

    /* finds clips with ownerOid <ownerOid> if <ownerOid> not null
     * and with tagTypes contained in <tagTypes> if specified
     * and tagBody = <tagBody> if <exact>
     * or tagBody like <tagBody> if not <exact>
     * at least one criterion must be specified.
     */
    public MediaClipTag[] findClips(String ownerOid, String[] tagTypes,
                                    String tagBody, boolean exact)
        throws RemoteException;
}
```

A.4 Channel classes

A.4.1 Channel business interface

```
package edu.mit.media.ic.mx.channel.business;

import edu.mit.media.ic.mx.businessobject.BusinessObject;

public interface Channel extends BusinessObject {

    public String getOwnerOid() throws RemoteException;
    public void setOwnerOid(String ownerOid) throws RemoteException;
    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public String getDescription() throws RemoteException;
    public void setDescription(String description) throws RemoteException;
    public boolean isOpen() throws RemoteException;
    public void setOpen(boolean bOpen) throws RemoteException;
    public boolean isHome() throws RemoteException;

}
```

A.4.2 ChannelController session bean

```
package edu.mit.media.ic.mx.channel.controller;

import edu.mit.media.ic.mx.channel.business.Channel;
import edu.mit.media.ic.mx.principal.business.User;
import edu.mit.media.ic.mx.util.*;

public interface ChannelController extends EJBObject {

    /* creates channel for user <owner> with specified
     * arguements
     */
    public Channel channelCreate(User owner, String name, String description,
                                 boolean open, boolean home)
        throws RemoteException;

    /* removes channel with OID <oid>
     */
    public void channelRemove(String oid)
        throws RemoteException;
```

```
    /* adds clip with OID <clipOid> to channel with OID <channelOid>
     * throws AccessFailedException if channel is private and <caller> not
     * owner
     * throws ObjectNotFoundException if channel not found
     */
    public void channelAddClip(User caller, String oid, String clipOid)
        throws RemoteException, ObjectNotFoundException, AccessFailedException;

    /* adds clip with OID <clipOid> to channel with OID <channelOid>
     * throws AccessFailedException if channel is private and <caller> not
     * owner
     * throws ObjectNotFoundException if channel not found
     */
    public void channelRemoveClip(User caller, String oid, String clipOid)
        throws RemoteException, ObjectNotFoundException, AccessFailedException;

    /* finds channel with OID <oid>
     * throws ObjectNotFoundException if channel not found
     */
    public Channel findChannelByOid(String oid)
        throws RemoteException, ObjectNotFoundException;

    /* finds home channel for user <caller>
     * throws ObjectNotFoundException if channel not found
     */
    public Channel findHomeChannel(User caller)
        throws RemoteException, ObjectNotFoundException;

    /* finds all channels owned by user <caller>
     * returns zero-length array if none found
     */
    public Channel[] findChannelsForUser(User caller)
        throws RemoteException;

    /* finds all channels owned with name <name>
     * returns zero-length array if none found
     */
    public Channel[] findChannelsByName(String name)
        throws RemoteException;
}
```

Appendix B

Library class specifications

B.1 Web-based connection libraries

B.1.1 MXWebUtil SMIL upload library class

```
package edu.mit.media.ic.lib;

public class MXWebUtil {

    /* initializes the utility class with the server name
     * and userOid
     */
    public static void init(String userOid, String server)

    /* uploads <smilString> and instructs the server to
     * create a SMIL file and clip with title <title>.
     * returns the OID of the SMIL media clip object,
     * which can then be used with the Clip class to
     * get clip metadata (URL, etc)
     * throws MXWebException in the event of an error.
     */
    public static String uploadSMILFile(String title, String smilString)
        throws MXWebException;

    /* updates the SMIL clip with OID <smilOid>,
     * setting its contents to <smilString>.
     * throws MXWebException in the event of an error.
     */
    public static void updateSMILFile(String smilOid, String smilString)
        throws MXWebException;
}
```

B.1.2 Clip class

```
package edu.mit.media.ic.lib;

public class Clip {

    /* creates a Clip object containing metadata
     * for media clip with OID <oid>
     */
    public Clip(String oid);

    /* returns false if clip does not exist
     */
    public boolean clipExists();
    public String getOid();
    public String getTitle();
    public String getDescription();
    public String getType();
    public String getOwner();
    public String getUrl();
    public int getFileSize();
    public boolean isLocal();

    /* returns a String representation of
     * Clip properties
     */
    public String toString();

    /* command line driver
     * takes one argument, a clip OID
     * creates a Clip object and prints information
     * to stdout
     */
    public static void main(String[] args);
}
```

B.2 Session bean connection library

B.2.1 MXControllerUtil library class

```
package edu.mit.media.ic.util;

public class MXControllerUtil {

    /* initializes library class with properties filename
     */
    public static void init(String filename)
```

```
throws RemoteException;

    /* gets java.lang.Object home object from JNDI name <homeName>
     * Class object (className.class) must be specified as second
     * argument.  Returned object must be case to home type before
     * use.
     */
    public static Object getHome(String homeName, Class classObj)
throws RemoteException {

Instance().lookup(homename, classObj);
    }
}
```

B.2.2 Sample properties file

This properties file contains the name of the server and JNDI names of objects. It can be customized by the client application.

```
SERVER_NAME=miyazaki.media.mit.edu
USER_CONTROLLER=mxEJB/UserController
MEDIACLIP_CONTROLLER=mxEJB/MediaClipController
TAG_CONTROLLER=mxEJB/TagController
CHANNEL_CONTROLLER=mxEJB/ChannelController
```

Appendix C

Server Hardware and Development Information

C.1 Hardware

The MediaExchange server currently supports database, application, web, and streaming media servers. In the future, the computational workload will be split up between at least two machines, one working as a database server and the other(s) working as the application server or application server cluster. Also, serving the streaming media can be offloaded to a dedicated machine for this purpose, with faster I/O and commercial streaming servers. This scenario is also likely, using the Media Lab's available servers for this purpose.

Currently the MediaExchange server hardware consists of a Pentium II machine running at 300 MHz; 384 megabytes of RAM; and 40+ GB of EIDE disk storage running off an Ultra-ATA/66 controller.

C.2 Development Environment

All the applications were developed using the standard command-line Java tools, javac, java, and jar. The Websphere jetace tool was used to package the EJBs with appropriate deployment descriptors, and the Websphere Administration Console was

used to monitor, control, and EJB-compile and deploy applications.

Oracle SQL+ was used for database administration.

Bibliography

[1] Java language to idl mapping: Object management group document ptc/99-03-09.

[2] Scott Ambler. Ambysoft white paper: Mapping objects to relational databases. 1999.

[3] J. D. Davidson and D. Coward. *Java Servlet 2.2. Spefication.* Sun Microsystems, Palo Alto, CA, 1999.

[4] Ejb!now website: `http://www.ejbnow.com`.

[5] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, first edition, 1994.

[6] V. Matena and M. Hapner. *Enterprise JavaBeans 1.1 Specification.* Sun Microsystems, Palo Alto, CA, 1999.

[7] Richard Monson-Haefel. *Enterprise Java Beans.* O'Reilly, Cambridge, Massachusetts, first edition, 1999.

[8] E. Pelegrí-Llopart and L. Cable. *Java Server Pages 1.1 Specification.* Sun Microsystems, Palo Alto, CA, 1999.

[9] Bill Venner. *Inside the Java Virtual Machine.* McGraw-Hill, New York, New York, first edition, 1998.

[10] Andreas Vogel and Keith Duddy. *Java Programming with CORBA*. John Wiley and Sons, New York, New York, second edition, 1998.

[11] Websphere applicaton server documentation: Writing enterprise beans in websphere, version 3.0.