1. INTRODUCTION

This project is aimed at developing a platform independent video or voice communication over Internet Protocol. Platform independent Video/Voice over Internet protocol is implemented by providing user interfaces at different terminals in an Ethernet. Each of these interfaces consists of components for providing textual, video and voice communication to the selected user currently logged to the system.

The main components of the project are following:

- List of users currently logged into the Ethernet
- Text Interfaces for sending text messages to selected user
- A voice-recording feature for recording and sending voice message.
- An interface for recording and playing audio file.
- A video viewing section to view video streams send from the connected user's web camera.

Besides the interface component the system requires a voice recording facility through a microphone and a video recording facility through a web camera.

1.1 Functionality Description:

Whenever we invoke the system, the list of users displays the list of all users currently logged to the Ethernet. It automatically includes the users who enter the network during out login period into the list. Selecting any of these users will invoke an interface designed for communication with the other users. The default mode of communication is text messaging. Activating other modes of communications like voice and video will result in a simultaneous communication with the other user in multiple modes.

2. SYSTEM STUDY

The First phase of software development is system study and analysis. The importance of system analysis phase is the establishment of the requirements for the system to be developed and installed. Analyzing the project to understand the complexity forms the vital part of the system study. Problematic areas are identified and information is collected. Fast finding or gathering is essential to any analysis of requirements. System analysis is the system approach to study and solution of problems using computer-based system. A system is an orderly grouping of independent components linked together according to a plan to achieve a specific objective. Each component is a part of the total system and has to do its share of work for the system to achieve the desired goal. This process is also referred to as life cycle methodology. Different stages of life cycle are:

Recognition of need: One must know what the objective is, before it can be achieved.

Feasibility study: A test of a system proposal according to its workability impact and orientation ability to meet user needs and effective use of resource.

Analysis: Analysis is a detailed study of various operations performed by the system and their relationships within and outside the system.

Design: Here we determine how the output is to be produced and in which format. Input data and master files have to be designed to meet the requirements of the proposed output.

Implementation: It is concerned with user training, site preparing and file conversion. During the final testing user acceptance is tested followed by user training.

System analysis includes investigation and possible changes to the existing system. Analysis is used to gain and understanding of the existing system and what is required of it. At the conclusion of the system analysis there is the system description and set of requirements for a new system. If there is no such existing system then analysis only defines the requirements. This new system may build afresh or by changing the existing system. Development begins by defining a model of the new system and continues this model to a working system. The model of the system shows what the system must do to satisfy these requirements. Finally data modes are converted to a database and processed to user procedures and computer programs

2.1 **Existing System**

Nowadays, people generally depend upon conventional means of mass information and entertainment like newspapers, television etc for their share of news, information and entertainment. These means are by far restrictive and the user can't exercise his own choice regarding the type news he wants or the sort of entertainment he likes. He might not always have access to critical information which might be urgently required for any activity of his. But with the penetration and awareness of E-connectivity the huge world of computer based infotainment is thrown open to him/her. The users can now exercise their own choice regarding what sort of information they want and readily access it. But they still have to find their stuff on the net. At present there are a very few portals which offer dedicated customer services. Most of them are of specialized nature and the user has to go hopping from one site to another to gather what they need in course of their daily life.

2.2 Limitations of Existing System

- At present there are a very few portals which offer dedicated customer services.
- The data and information provided are not regularly updated and hence might not be totally reliable
- Also most of the sites are of specialized. Nature and user has to go hopping from one site to another to gather what they need in course of their daily life.
- The information is spread all over the cyberspace and it often places the user in a dilemma as to where to find what he wants.

2.3 The proposed system

In the proposed system we plane to bring the services provided by a range of websites under a single window, thereby saving the user the time and effort required to search around at numerous sites. Initially the service are planned to be of basic nature which will cater to the minimum needs of an individual. The portal under consideration will provide dedicated services in some selected fields to the customer. The data and information provided would be regularly updated and hence the customer need not worry about the accuracy and reliability of the data. The portal is proposed to be of general nature and the visitors can find all information of daily interest under a single roof. This will save a whole lot of their time and effort. . Considering the hectic lifestyle which most people have to keep up with in today's world, saving of time and avoidable effort would be most welcome.

3. SYSTEM REQUIREMENTS

3.1 Hardware requirements:

Processor	:	Pentium IV 850 MHz or above
Main Memory	:	128 MB RAM
Hard Disk	:	20 GB
Network Interface Card	:	100 MB Ethernet Card
I/O Devices	:	Web Cam, Head Phone, Keyboard, Mouse, VGA compatible Monitor

3.2 Software requirements:

The proposed system will be functional over Internet Protocol implemented Ethernets. The functionality will be developed using Java 2 standard edition 1.4 and with java packages like Java Media Framework (JMF) etc.

Language : JAVA

Platform : Windows XP

4. SOFTWARE DESCRIPTION

4.1 JAVA

Java was developed at Sun Microsystems. Work on Java initially began with the goal of creating a platform-independent language and OS for consumer electronics. The original intent was to use C++, but as work progressed in this direction, developers identified that creating their own language would serve them better. The effort towards consumer electronics led the Java team, then known as First Person Inc., towards developing h/w and s/w for the delivery of video-on-demand with Time Warner.

Unfortunately (or fortunately for us) Time Warner selected Silicon Graphics as the vendor for video-on-demand project. This set back left the First Person team with an interesting piece of s/w (Java) and no market to place it. Eventually, the natural synergies of the Java language and the www were noticed, and Java found a market.

Today Java is both a programming language and an environment for executing programs written in Java Language. Unlike traditional compilers, which convert source code into machine level instructions, the Java compiler translates java source code into instructions that are interpreted by the runtime Java Virtual Machine. So unlike languages like C and C++, on which Java is based, Java is an interpreted language.

Java is the first programming language designed from ground up with network programming in mind. The core API for Java includes classes and interfaces that provide uniform access to a diverse set of network protocols. As the Internet and network programming have evolved, Java has maintained its cadence. New APIs and toolkits have expanded the available options for the Java network programmer.

Why Is Java Interesting?

In one of their early papers about the language, Sun described Java as follows: Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.
Sun acknowledges that this is quite a string of buzzwords, but the fact is that, for the most part, they aptly describe the language. In order to understand why Java is so interesting, let's take a look at the language features behind the buzzwords.

Object-Oriented

Java is an *object-oriented* programming language. As a programmer, this means that you focus on the data in your application and methods that manipulate that data, rather than thinking strictly in terms of procedures. If you're accustomed to procedure-based programming in C, you may find that you need to change how you design your programs when you use Java. Once you see how powerful this new paradigm is, however, you'll quickly adjust to it. In an object-oriented system, a *class* is a collection of data and methods that operate on that data. Taken together, the data and methods describe the state and behavior of an *object*. Classes are arranged in a hierarchy, so that a subclass can inherit behavior from its superclass. A class hierarchy always has a root class; this is a class with very general behavior. Java comes with an extensive set of classes, arranged in *packages*, that you can use in your programs. For example, Java provides classes that create graphical user interface components (the java.awt package), classes that handle input and output (the java.io package), and classes that support networking functionality (the java.net package). The Object class (in the java.lang package) serves as the root of the Java class hierarchy. Unlike C++, Java was designed to be object-oriented from the ground up. Most things in Java are objects; the primitive numeric, character, and boolean types are the only exceptions. Strings are represented by objects in Java, as are other important language constructs like threads. A class is the basic unit of compilation and of execution in Java; all Java programs are classes. While Java is designed to look like C++,

you'll find that Java removes many of the complexities of that language. If you are a C++ programmer, you'll want to study the object-oriented constructs in Java carefully. Although the syntax is often similar to C++, the behavior is not nearly so analogous. For a complete description of the object-oriented features of Java, The object oriented language used to create executable contents such as applications and applets.

Interpreted

Java is an interpreted language: the Java compiler generates byte-codes for the Java Virtual Machine (JVM), rather than native machine code. To actually run a Java program, you use the Java interpreter to execute the compiled byte-codes. Because Java byte-codes are platform-independent, Java programs can run on any platform that the JVM (the interpreter and run-time system) has been ported to. In an interpreted environment, the standard "link" phase of program development pretty much vanishes. If Java has a link phase at all, it is only the process of loading new classes into the environment, which is an incremental, lightweight process that occurs at run-time. This is in contrast with the slower and more cumbersome compile-link-run cycle of languages like C and C++.

Architecture Neutral and Portable

Because Java programs are compiled to an *architecture neutral* byte-code format, a Java application can run on any system, as long as that system implements the Java Virtual Machine. This is a particularly important for applications distributed over the Internet or other heterogeneous networks. But the architecture neutral approach is useful beyond the scope of network-based applications. As an application developer in today's software market, you probably want to develop versions of your application that can run on PCs, Macs, and UNIX workstations. With multiple flavors of UNIX, Windows 95, and Windows NT on the PC, and the new PowerPC Macintosh, it is becoming increasingly difficult to produce software for all of the possible platforms. If you write your application in Java, however, it can run on all platforms. The fact that Java is interpreted and defines a standard, architecture neutral, byte-code format is one big part of being *portable*. But Java goes even further, by making sure that there are no "implementation-dependent" aspects of the language specification. For example, Java explicitly specifies

the size of each of the primitive data types, as well as its arithmetic behavior. This differs from C, for example, in which an int type can be 16, 32, or 64 bits long depending on the platform. While it is technically possible to write non-portable programs in Java, it is relatively easy to avoid the few platform-dependencies that are exposed by the Java API and write truly portable or "pure" Java programs. Sun's new "100% Pure Java" program helps developers ensure (and certify) that their code is portable. Programmers need only to make simple efforts to avoid non-portable pitfalls in order to live up to Sun's trademarked motto "Write Once, Run Anywhere."

Dynamic and Distributed

Java is a *dynamic* language. Any Java class can be loaded into a running Java interpreter at any time. These dynamically loaded classes can then be dynamically instantiated. Native code libraries can also be dynamically loaded. Classes in Java are represented by the Class class; you can dynamically obtain information about a class at run-time. This is especially true in Java 1.1, with the addition of the Reflection API. Java is also called a *distributed* language. This means, simply, that it provides a lot of high-level support for networking. For example, the URL class and OArelated classes in the java.net package make it almost as easy to read a remote file or resource as it is to read a local file. Similarly, in Java 1.1, the Remote Method Invocation (RMI) API allows a Java program to invoke methods of remote Java objects, as if they were local objects. (Java also provides traditional lower-level networking support, including datagrams and stream-based connections through sockets.) The distributed nature of Java really shines when combined with its dynamic class loading capabilities. Together, these features make it possible for a Java interpreter to download and run code from across the Internet. (As we'll see below, Java implements strong security measures to be sure that this can be done safely.) This is what happens when a Web browser downloads and runs a Java applet, for example. Scenarios can be more complicated than this, however. Imagine a multi-media word processor written in Java. When this program is asked to display some type of data that it has never encountered before, it might dynamically download a class from the network that can parse the data, and then dynamically download another class (probably a Java "bean") that can display the data within a compound document. A program like this

uses distributed resources on the network to dynamically grow and adapt to the needs of its user.

Simple

Java is a *simple* language. The Java designers were trying to create a language that a programmer could learn quickly, so the number of language constructs has been kept relatively small. Another design goal was to make the language look familiar to a majority of programmers, for ease of migration. If you are a C or C++ programmer, you'll find that Java uses many of the same language constructs as C and C++. In order to keep the language both small and familiar, the Java designers removed a number of features available in C and C++. These features are mostly ones that led to poor programming practices or were rarely used. For example, Java does not support the goto statement; instead, it provides labeled break and continue statements and exception handling. Java does not use header files and it eliminates the C preprocessor. Because Java is object-oriented, C constructs like struct and union have been removed. Java also eliminates the operator overloading and multiple inheritance features of C++. Perhaps the most important simplification, however, is that Java does not use pointers. Pointers are one of the most bug-prone aspects of C and C++ programming. Since Java does not have structures, and arrays and strings are objects, there's no need for pointers. Java automatically handles the referencing and dereferencing of objects for you. Java also implements automatic garbage collection, so you don't have to worry about memory management issues. All of this frees you from having to worry about dangling pointers, invalid pointer references, and memory leaks, so you can spend your time developing the functionality of your programs. If it sounds like Java has gutted C and C++, leaving only a shell of a programming language, hold off on that judgment for a bit, Java is actually a full-featured and very elegant language.

Robust

Java has been designed for writing highly reliable or *robust* software. Java certainly doesn't eliminate the need for software quality assurance; it's still quite possible to write buggy software in Java. However, Java does eliminate certain types of programming errors, which makes it considerably easier to write reliable software. Java is a strongly

typed language, which allows for extensive compile-time checking for potential type-mismatch problems. Java is more strongly typed than C++, which inherits a number of compile-time laxities from C, especially in the area of function declarations. Java requires explicit method declarations; it does not support C-style implicit declarations. These stringent requirements ensure that the compiler can catch method invocation errors, which leads to more reliable programs. One of the things that makes Java simple is its lack of pointers and pointer arithmetic. This feature also increases the robustness of Java programs by abolishing an entire class of pointer-related bugs. Similarly, all accesses to arrays and strings are checked at run-time to ensure that they are in bounds, eliminating the possibility of overwriting memory and corrupting data. Casts of objects from one type to another are also checked at run-time to ensure that they are legal. Finally, and very importantly, Java's automatic garbage collection prevents memory leaks and other pernicious bugs related to memory allocation and deallocation. Exception handling is another feature in Java that makes for more robust programs. An *exception* is a signal that some sort of exceptional condition, such as a "file not found" error, has occurred. Using the try/catch/finally statement, you can group all of your error handling code in one place, which greatly simplifies the task of error handling and recovery.

Secure

One of the most highly touted aspects of Java is that it's a *secure* language. This is especially important because of the distributed nature of Java. Without an assurance of security, you certainly wouldn't want to download code from a random site on the Internet and let it run on your computer. Yet this is exactly what people do with Java applets every day. Java was designed with security in mind, and provides several layers of security controls that protect against malicious code, and allow users to comfortably run untrusted programs such as applets. At the lowest level, security goes hand-in-hand with robustness. As we've already seen, Java programs cannot forge pointers to memory, or overflow arrays, or read memory outside of the bounds of an array or string. These features are one of Java's main defenses against malicious code. By totally disallowing any direct access to memory, an entire huge, messy class of security attacks is ruled out.

The second line of defense against malicious code is the byte-code verification process that the Java interpreter performs on any untrusted code it loads. These verification steps ensure that the code is well-formed--that it doesn't overflow or underflow the stack or contain illegal byte-codes, for example. If the byte-code verification step was skipped, inadvertently corrupted or maliciously crafted byte-codes might be able to take advantage of implementation weaknesses in a Java interpreter. Another layer of security protection is commonly referred to as the "sandbox model": untrusted code is placed in a "sandbox," where it can play safely, without doing any damage to the "real world," or full Java environment. When an applet, or other untrusted code, is running in the sandbox, there are a number of restrictions on what it can do. The most obvious of these restrictions is that it has no access whatsoever to the local file system. There are a number of other restrictions in the sandbox as well. These restrictions are enforced by a Security Manager class. The model works because all of the core Java classes that perform sensitive operations, such as file system access, first ask permission of the currently installed Security Manager. If the call is being made, directly or indirectly, by untrusted code, the security manager throws an exception, and the operation is not permitted. Finally, in Java 1.1, there is another possible solution to the problem of security. By attaching a digital signature to Java code, the origin of that code can be established in a cryptographically secure and unforgeable way. If you have specified that you trust a person or organization, then code that bears the digital signature of that trusted entity is trusted, even when loaded over the network, and may be run without the restrictions of the sandbox model. Of course, security isn't a black-and-white thing. Just as a program can never be guaranteed to be 100% bug-free, no language or environment can be guaranteed 100% secure. With that said, however, Java does seem to offer a practical level of security for most applications. It anticipates and defends against most of the techniques that have historically been used to trick software into misbehaving, and it has been intensely scrutinized by security experts and hackers alike. Some security holes were found in early versions of Java, but these flaws were fixed almost as soon as they were found, and it seems reasonable to expect that any future holes will be fixed just as quickly.

High-Performance

Java is an interpreted language, so it is never going to be as fast as a compiled language like C. Java 1.0 was said to be about 20 times slower than C. Java 1.1 is nearly twice as fast as Java 1.0, however, so it might be reasonable to say that compiled C code runs ten times as fast as interpreted Java byte-codes. But before you throw up your arms in disgust, be aware that this speed is more than adequate to run interactive, GUI and network-based applications, where the application is often idle, waiting for the user to do something, or waiting for data from the network. Furthermore, the speed-critical sections of the Java run-time environment, that do things like string concatenation and comparison, are implemented with efficient native code. As a further performance boost, many Java interpreters now include "just in time" compilers that can translate Java byte-codes into machine code for a particular CPU at run-time. The Java byte-code format was designed with these "just in time" compilers in mind, so the process of generating machine code is fairly efficient and it produces reasonably good code. In fact, Sun claims that the performance of byte-codes converted to machine code is nearly as good as native C or C++. If you are willing to sacrifice code portability to gain speed, you can also write portions of your program in C or C++ and use Java native methods to interface with this native code. When you are considering performance, it's important to remember where Java falls in the spectrum of available programming languages. At one end of the spectrum, there are high-level, fully-interpreted scripting languages such as Tcl and the UNIX shells. These languages are great for prototyping and they are highly portable, but they are also very slow. At the other end of the spectrum, you have low-level compiled languages like C and C++. These languages offer high performance, but they suffer in terms of reliability and portability. Java falls in the middle of the spectrum. The performance of Java's interpreted byte-codes is much better than the high-level scripting languages (even Perl), but it still offers the simplicity and portability of those languages.

Multithreaded

In a GUI-based network application such as a Web browser, it's easy to imagine multiple things going on at the same time. A user could be listening to an audio clip while she is scrolling a page, and in the background the browser is downloading an image. Java is a *multithreaded* language; it provides support for multiple threads of execution (sometimes

called lightweight processes) that can handle different tasks. An important benefit of multithreading is that it improves the interactive performance of graphical applications for the user. If you have tried working with threads in C or C++, you know that it can be quite difficult. Java makes programming with threads much easier, by providing built-in anguage support for threads. The java.lang package provides a Thread class that supports methods to start and stop threads and set thread priorities, among other things. The Java language syntax also supports threads directly with the synchronized keyword. This keyword makes it extremely easy to mark sections of code or entire methods that should only be run by a single thread at a time. While threads are "wizard-level" stuff in C and C++, their use is commonplace in Java. Because Java makes threads so easy to use, the Java class libraries require their use in a number of places. For example, any applet that performs animation does so with a thread. Similarly, Java does not support asynchronous, non-blocking I/O with notification through signals or interrupts--you must instead create a thread that blocks on every I/O channel you are interested in.

Java Runtime Environment

The runtime environment used to execute the code. It is made up of the java language and java virtual machine. It is portable and it is platform neutral.

Java tools

It is used by the developers to create java code. They include java compiler, java interpreter, classes, libraries and applet viewer.

Java Application

Applications are programs written in java to carry out certain tasks on stand alone local computer. Execution of a stand alone program involves two steps.

Compiling the source code into byte code using javac.

Executing byte code program using java interpreter.

Java Applets

Java applets are pieces of java code that are embedded in HTML document using the applet tag. When the browser encounters such code it automatically download it and execute it.

Java Virtual Machine

It is a specification to which java codes must be written. All java code is to be compiled to be used in this nonexistent virtual machine. Writing the code which compiles in JVM ensures platform independence.

ADVANTAGES OF JAVA

Java is Robust

Robust programs are those reliable programs which are unlikely to fail even under the most unlikely conditions. Many languages like C do not have this feature because they are relaxed in terms of type checking in terms of programming errors. Java is strict about type declarations and does not allow automatic typecasting. Also it uses a pointer model that does not overwrite memory or corrupt data.

Java is secure

Java allows creation of virus-free, tamper free systems to be created. It ensures security in the following ways.

Pointers and memory allocations are removed during compile time.

All byte codes are verified by the interpreter before executing.

All Java applets are treated as untrusted code executing in trusted environment.

Because Java was written to support distributed applications over the computer networks, it can be used with a variety of CPU and operating system architectures. To achieve this goal a compiler was created that produces architecture-neutral object files from Java code.

Java is portable

Java byte code will be executed on any computer that has Java run time environment. The portability is achieved in the following ways.

Java primitive data types and the behavior of arithmetic operations on these data types are explicitly specified.

The Java libraries include portable interfaces for each platform on which the run time environment is available.

The entire Java system itself is portable.

Java is small

Because java was designed to run on small computers, java system is relatively small for a programming language. It can run efficiently on PCs with 4 MB RAM or more. The java interpreter takes up only a few hundred kilo bytes.

Java is garbage collected

Java programs don't have to worry about memory management. The Java system has a built in program called the garbage collector, which scans the memory and automatically frees the memory chunks that are not in use.

Java is dynamic

Fundamentally distributed computer environments must be dynamic. Java is capable of dynamically linking new libraries, methods and instance variables as it goes without breaking and without concern.

4.2 J2EE

Distributed Multitiered Applications

The J2EE platform uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the various application components that make up a J2EE application are installed on different machines depending on the tier in the multitiered J2EE environment to which the application component belongs. Figure 1-1 shows two multitiered J2EE applications divided into the tiers described in the following list. The J2EE application parts shown in Figure 1-1 are presented in J2EE Components.

- Client-tier components run on the client machine.
- Web-tier components run on the J2EE server.
- Business-tier components run on the J2EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Although a J2EE application can consist of the three or four tiers shown in Figure 1-1, J2EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the J2EE server machine, and the database or legacy machines at the back end. Three-tiered applications that run in this way extend the standard two-tiered client and server model by placing a multithreaded application server between the client application and back-end storage.

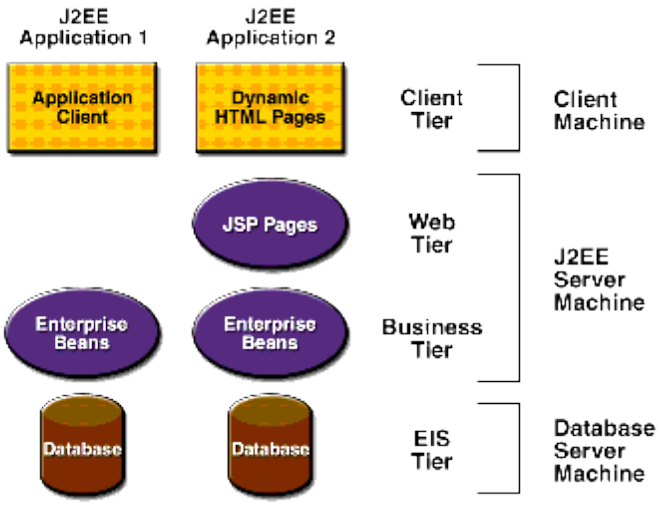


Figure 1-1 Multitiered Applications

J2EE Components

J2EE applications are made up of components. A *J2EE component* is a self-contained functional software unit that is assembled into a J2EE application with its related classes and files and that communicates with other components. The J2EE specification defines the following J2EE components:

- Application clients and applets are components that run on the client.
- Java Servlet and JavaServer Pages™ (JSP™) technology components are web components that run on the server.
- Enterprise JavaBeans™ (EJB™) components (enterprise beans) are business components that run on the server.

J2EE components are written in the Java programming language and are compiled in the same way as any program in the language. The difference between J2EE components and "standard" Java classes is that J2EE components are assembled into a J2EE application, are verified to be well formed and in compliance with the J2EE specification, and are deployed to production, where they are run and managed by the J2EE server.

J2EE Clients

A J2EE client can be a web client or an application client.

Web Clients

A *web client* consists of two parts: (1) dynamic web pages containing various types of markup language (HTML, XML, and so on), which are generated by web components running in the web tier, and (2) a web browser, which renders the pages received from the server.

A web client is sometimes called a *thin client*. Thin clients usually do not query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, such heavyweight operations are off-loaded to enterprise beans executing on the J2EE server, where they can leverage the security, speed, services, and reliability of J2EE server-side technologies.

Applets

A web page received from the web tier can include an embedded applet. An *applet* is a small client application written in the Java programming language that executes in the Java virtual machine installed in the web browser. However, client systems will likely need the Java Plug-in and possibly a security policy file in order for the applet to successfully execute in the web browser.

Web components are the preferred API for creating a web client program because no plug-ins or security policy files are needed on the client systems. Also, web components enable cleaner and more modular application design because they provide a way to

separate applications programming from web page design. Personnel involved in web page design thus do not need to understand Java programming language syntax to do their jobs.

Application Clients

An *application client* runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from the Swing or the Abstract Window Toolkit (AWT) API, but a command-line interface is certainly possible.

Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a servlet running in the web tier.

The JavaBeans™ Component Architecture

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans components) to manage the data flow between an application client or applet and components running on the J2EE server, or between server components and a database. JavaBeans components are not considered J2EE components by the J2EE specification.

JavaBeans components have properties and have get and set methods for accessing the properties. JavaBeans components used in this way are typically simple in design and implementation but should conform to the naming and design conventions outlined in the JavaBeans component architecture.

J2EE Server Communications

Figure 1-2 shows the various elements that can make up the client tier. The client communicates with the business tier running on the J2EE server either directly or, as in the case of a client running in a browser, by going through JSP pages or servlets running in the web tier.

Your J2EE application uses a thin browser-based client or thick application client. In deciding which one to use, you should be aware of the trade-offs between keeping functionality on the client and close to the user (thick client) and off-loading as much functionality as possible to the server (thin client). The more functionality you off-load to the server, the easier it is to distribute, deploy, and manage the application; however, keeping more functionality on the client can make for a better perceived user experience.

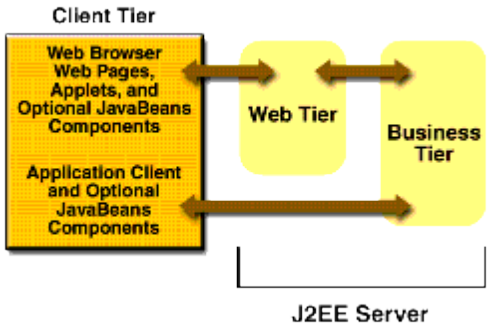


Figure 1-2 Server Communications

Web Components

J2EE web components are either servlets or pages created using JSP technology (JSP pages). *Servlets* are Java programming language classes that dynamically process requests and construct responses. *JSP pages* are text-based documents that execute as servlets but allow a more natural approach to creating static content.

Static HTML pages and applets are bundled with web components during application assembly but are not considered web components by the J2EE specification. Server-side utility classes can also be bundled with web components and, like HTML pages, are not considered web components.

As shown in Figure 1-3, the web tier, like the client tier, might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.

Business Components

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in the business tier. Figure 1-4 shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enterprise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.

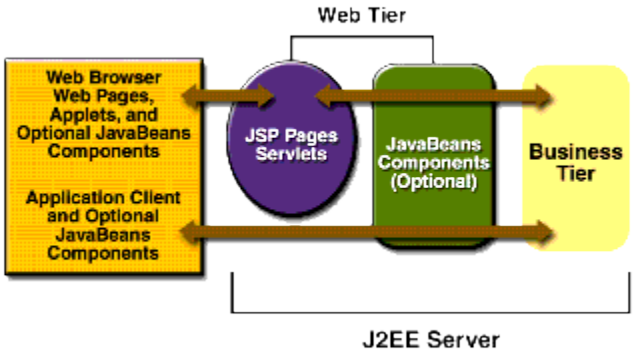


Figure 1-3 Web Tier and J2EE Applications

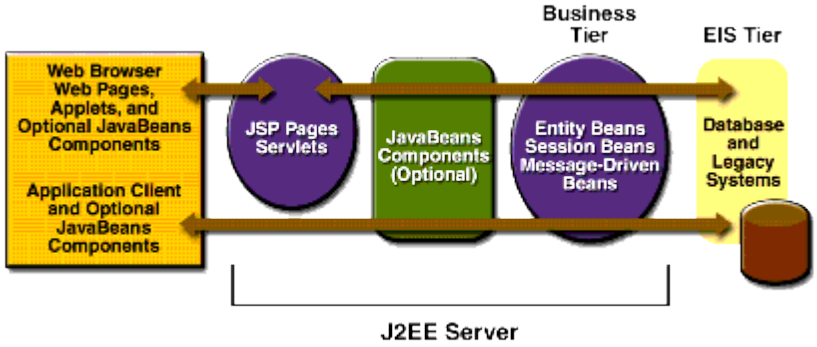


Figure 1-4 Business and EIS Tiers

There are three kinds of enterprise beans: session beans, entity beans, and message-driven beans. A *session bean* represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone. In contrast, an *entity bean* represents persistent data stored in one row of a database table. If the client terminates or

if the server shuts down, the underlying services ensure that the entity bean data is saved. A *message-driven bean* combines features of a session bean and a Java Message Service (JMS) message listener, allowing a business component to receive JMS messages asynchronously.

Enterprise Information System Tier

The enterprise information system tier handles EIS software and includes enterprise infrastructure systems such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. For example, J2EE application components might need access to enterprise information systems for database connectivity.

J2EE Containers

Normally, thin-client multitiered applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details. The component-based and platform-independent J2EE architecture makes J2EE applications easy to write because business logic is organized into reusable components. In addition, the J2EE server provides underlying services in the form of a container for every component type. Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.

Container Services

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Before a web component, enterprise bean, or application client component can be executed, it must be assembled into a J2EE module and deployed into its container.

The assembly process involves specifying container settings for each component in the J2EE application and for the J2EE application itself. Container settings customize the underlying support provided by the J2EE server, including services such as security, transaction management, Java Naming and Directory Interface™ (JNDI) lookups, and remote connectivity. Here are some of the highlights:

- The J2EE security model lets you configure a web component or enterprise bean so that system resources are accessed only by authorized users.
- The J2EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access naming and directory services.
- The J2EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

Because the J2EE architecture provides configurable services, application components within the same J2EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages nonconfigurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the J2EE platform APIs. Although data persistence is a nonconfigurable service, the J2EE architecture lets you override container-managed persistence by including the appropriate code in your enterprise bean implementation when you want more control than the default container-managed persistence provides. For example, you might use bean-managed persistence to implement your own finder (search) methods or to create a customized database cache.

Container Types

The deployment process installs J2EE application components in the J2EE containers illustrated in Figure 1-5.

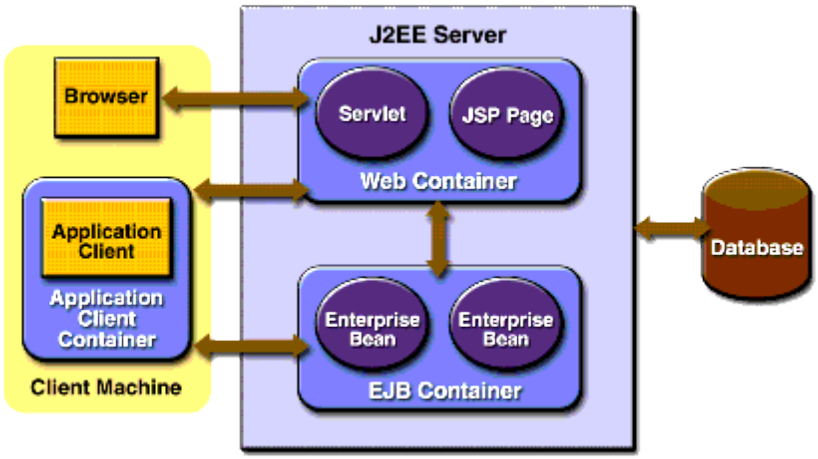


Figure 1-5 J2EE Server and Containers

J2EE server

The runtime portion of a J2EE product. A J2EE server provides EJB and web containers.

Enterprise JavaBeans (EJB) container

Manages the execution of enterprise beans for J2EE applications. Enterprise beans and their container run on the J2EE server.

Web container

Manages the execution of JSP page and servlet components for J2EE applications. Web components and their container run on the J2EE server.

Application client container

Manages the execution of application client components. Application clients and their container run on the client.

Applet container

Manages the execution of applets. Consists of a web browser and Java Plug-in running on the client together.

Web Services Support

Web services are web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with calling clients. The J2EE platform provides the XML APIs and tools you need to quickly design, develop, test, and deploy web services and clients that fully interoperate with other web services and clients running on Java-based or non-Java-based platforms.

To write web services and clients with the J2EE XML APIs, all you do is pass parameter data to the method calls and process the data returned; or for document-oriented web services, you send documents containing the service data back and forth. No low-level programming is needed because the XML API implementations do the work of translating the application data to and from an XML-based data stream that is sent over the standardized XML-based transport protocols. These XML-based standards and protocols are introduced in the following sections.

The translation of data to a standardized XML-based data stream is what makes web services and clients written with the J2EE XML APIs fully interoperable. This does not necessarily mean that the data being transported includes XML tags because the transported data can itself be plain text, XML data, or any kind of binary data such as audio, video, maps, program files, computer-aided design (CAD) documents and the like.

The next section introduces XML and explains how parties doing business can use XML tags and schemas to exchange data in a meaningful way.

XML

XML is a cross-platform, extensible, text-based standard for representing data. When XML data is exchanged between parties, the parties are free to create their own tags to describe the data, set up schemas to specify which tags can be used in a particular kind of XML document, and use XML stylesheets to manage the display and handling of the data.

For example, a web service can use XML and a schema to produce price lists, and companies that receive the price lists and schema can have their own stylesheets to handle the data in a way that best suits their needs. Here are examples:

- One company might put XML pricing information through a program to translate the XML to HTML so that it can post the price lists to its intranet.
- A partner company might put the XML pricing information through a tool to create a marketing presentation.
- Another company might read the XML pricing information into an application for processing.

SOAP Transport Protocol

Client requests and web service responses are transmitted as Simple Object Access Protocol (SOAP) messages over HTTP to enable a completely interoperable exchange between clients and web services, all running on different platforms and at various locations on the Internet. HTTP is a familiar request-and response standard for sending messages over the Internet, and SOAP is an XML-based protocol that follows the HTTP request-and-response model.

The SOAP portion of a transported message handles the following:

- Defines an XML-based envelope to describe what is in the message and how to process the message
- Includes XML-based encoding rules to express instances of application-defined data types within the message
- Defines an XML-based convention for representing the request to the remote service and the resulting response

WSDL Standard Format

The Web Services Description Language (WSDL) is a standardized XML format for describing network services. The description includes the name of the service, the location of the service, and ways to communicate with the service. WSDL service descriptions can be stored in registries or published on the web (or both). The Sun Java System Application Server Platform Edition 8 provides a tool for generating the WSDL specification of a web service that uses remote procedure calls to communicate with clients.

UDDI and ebXML Standard Formats

Other XML-based standards, such as Universal Description, Discovery and Integration (UDDI) and ebXML, make it possible for businesses to publish information on the Internet about their products and web services, where the information can be readily and globally accessed by clients who want to do business.

Packaging Applications

A J2EE application is delivered in an Enterprise Archive (EAR) file, a standard Java Archive (JAR) file with an .ear extension. Using EAR files and modules makes it possible to assemble a number of different J2EE applications using some of the same components. No extra coding is needed; it is only a matter of assembling (or packaging) various J2EE modules into J2EE EAR files.

An EAR file (see Figure 1-6) contains J2EE modules and deployment descriptors. A *deployment descriptor* is an XML document with an .xml extension that describes the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the J2EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

There are two types of deployment descriptors: J2EE and runtime. A *J2EE deployment descriptor* is defined by a J2EE specification and can be used to configure deployment settings on any J2EE-compliant implementation. A *runtime deployment descriptor* is used to configure J2EE implementation-specific parameters. For example, the Sun Java System Application Server Platform Edition 8 runtime deployment descriptor contains information such as the context root of a web application, the mapping of portable names of an application's resources to the server's resources, and Application Server implementation-specific parameters, such as caching directives. The Application Server runtime deployment descriptors are named sun-*moduleType*.xml and are located in the same directory as the J2EE deployment descriptor.

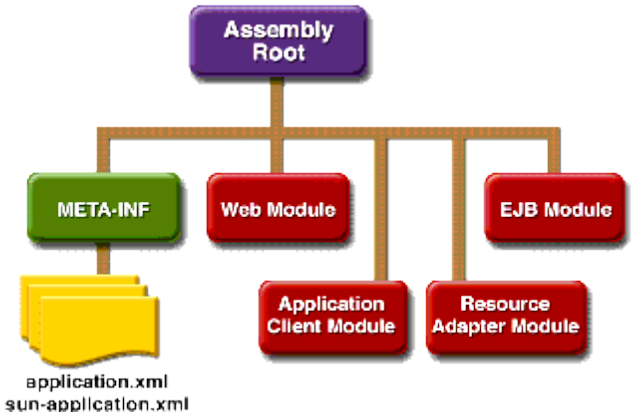


Figure 1-6 EAR File Structure

A *J2EE module* consists of one or more J2EE components for the same container type and one component deployment descriptor of that type. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. A J2EE module without an application deployment

descriptor can be deployed as a *stand-alone* module. The four types of J2EE modules are as follows:

- EJB modules, which contain class files for enterprise beans and an EJB deployment descriptor. EJB modules are packaged as JAR files with a .jar extension.
- Web modules, which contain servlet class files, JSP files, supporting class files, GIF and HTML files, and a web application deployment descriptor. Web modules are packaged as JAR files with a .war (web archive) extension.
- Application client modules, which contain class files and an application client deployment descriptor. Application client modules are packaged as JAR files with a .jar extension.
- Resource adapter modules, which contain all Java interfaces, classes, native libraries, and other documentation, along with the resource adapter deployment descriptor. Together, these implement the Connector architecture (see J2EE Connector Architecture) for a particular EIS. Resource adapter modules are packaged as JAR files with an .rar (resource adapter archive) extension.

Development Roles

Reusable modules make it possible to divide the application development and deployment process into distinct roles so that different people or companies can perform different parts of the process.

The first two roles involve purchasing and installing the J2EE product and tools. After software is purchased and installed, J2EE components can be developed by application component providers, assembled by application assemblers, and deployed by application deployers. In a large organization, each of these roles might be executed by different individuals or teams. This division of labor works because each of the earlier roles outputs a portable file that is the input for a subsequent role. For example, in the

application component development phase, an enterprise bean software developer delivers EJB JAR files. In the application assembly role, another developer combines these EJB JAR files into a J2EE application and saves it in an EAR file. In the application deployment role, a system administrator at the customer site uses the EAR file to install the J2EE application into a J2EE server.

The different roles are not always executed by different people. If you work for a small company, for example, or if you are prototyping a sample application, you might perform the tasks in every phase.

J2EE Product Provider

The J2EE product provider is the company that designs and makes available for purchase the J2EE platform APIs, and other features defined in the J2EE specification. Product providers are typically operating system, database system, application server, or web server vendors who implement the J2EE platform according to the Java 2 Platform, Enterprise Edition specification.

Tool Provider

The tool provider is the company or person who creates development, assembly, and packaging tools used by component providers, assemblers, and deployers.

Application Component Provider

The application component provider is the company or person who creates web components, enterprise beans, applets, or application clients for use in J2EE applications.

Enterprise Bean Developer

An enterprise bean developer performs the following tasks to deliver an EJB JAR file that contains the enterprise bean(s):

- Writes and compiles the source code

- Specifies the deployment descriptor
- Packages the .class files and deployment descriptor into the EJB JAR file

Web Component Developer

A web component developer performs the following tasks to deliver a WAR file containing the web component(s):

- Writes and compiles servlet source code
- Writes JSP and HTML files
- Specifies the deployment descriptor
- Packages the .class, .jsp, and.html files and deployment descriptor into the WAR file

Application Client Developer

An application client developer performs the following tasks to deliver a JAR file containing the application client:

- Writes and compiles the source code
- Specifies the deployment descriptor for the client
- Packages the .class files and deployment descriptor into the JAR file

Application Assembler

The application assembler is the company or person who receives application modules from component providers and assembles them into a J2EE application EAR file. The assembler or deployer can edit the deployment descriptor directly or can use tools that correctly add XML tags according to interactive selections. A software developer performs the following tasks to deliver an EAR file containing the J2EE application:

- Assembles EJB JAR and WAR files created in the previous phases into a J2EE application (EAR) file
- Specifies the deployment descriptor for the J2EE application

- Verifies that the contents of the EAR file are well formed and comply with the J2EE specification

Application Deployer and Administrator

The application deployer and administrator is the company or person who configures and deploys the J2EE application, administers the computing and networking infrastructure where J2EE applications run, and oversees the runtime environment. Duties include such things as setting transaction controls and security attributes and specifying connections to databases.

During configuration, the deployer follows instructions supplied by the application component provider to resolve external dependencies, specify security settings, and assign transaction attributes. During installation, the deployer moves the application components to the server and generates the container-specific classes and interfaces.

A deployer or system administrator performs the following tasks to install and configure a J2EE application:

- Adds the J2EE application (EAR) file created in the preceding phase to the J2EE server
- Configures the J2EE application for the operational environment by modifying the deployment descriptor of the J2EE application
- Verifies that the contents of the EAR file are well formed and comply with the J2EE specification
- Deploys (installs) the J2EE application EAR file into the J2EE server

J2EE 1.4 APIs

Figure 1-7 illustrates the availability of the J2EE 1.4 platform APIs in each J2EE container type. The following sections give a brief summary of the technologies required by the J2EE platform and the J2SE enterprise APIs that would be used in J2EE applications.

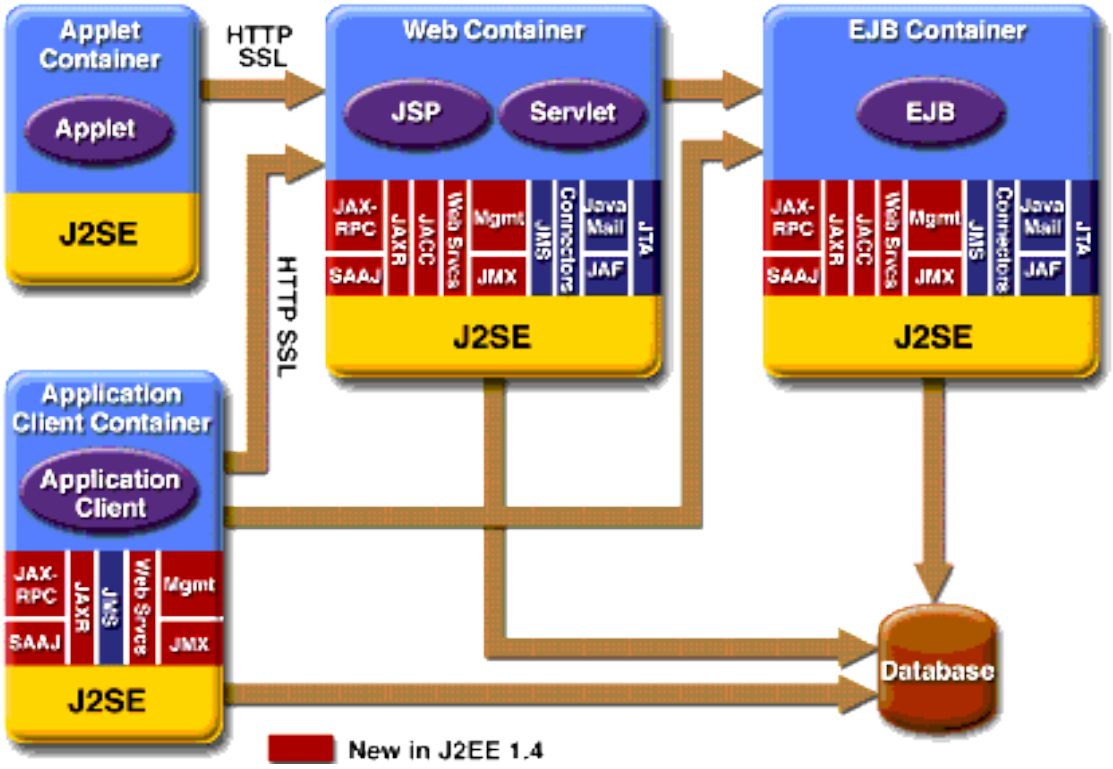


Figure 1-7 J2EE Platform APIs

Enterprise JavaBeans Technology

An Enterprise JavaBeans™ (EJB™) component, or *enterprise bean*, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the J2EE server.

As mentioned earlier, there are three kinds of enterprise beans: session beans, entity beans, and message-driven beans. Enterprise beans often interact with databases. One of the benefits of entity beans is that you do not have to write any SQL code or use the JDBC™ API (see JDBC API) directly to perform database access operations; the EJB container handles this for you. However, if you override the default container-managed persistence for any reason, you will need to use the JDBC API. Also, if you choose to have a session bean access the database, you must use the JDBC API.

Java Servlet Technology

Java servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications that are accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

JavaServer Pages Technology

JavaServer Pages™ (JSP™) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text: static data (which can be expressed in any text-based format such as HTML, WML, and XML) and JSP elements, which determine how the page constructs dynamic content.

Java Message Service API

The Java Message Service (JMS) API is a messaging standard that allows J2EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.

Java Transaction API

The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The J2EE architecture provides a default auto commit to handle transaction commits and rollbacks. An *auto commit* means that any other applications that are viewing data will see the updated data after each database read or write operation. However, if your application performs two separate database access operations that depend on each other, you will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

JavaMail API

J2EE applications use the JavaMail™ API to send email notifications. The JavaMail API has two parts: an application-level interface used by the application components to send

mail, and a service provider interface. The J2EE platform includes JavaMail with a service provider that allows application components to send Internet mail.

JavaBeans Activation Framework

The JavaBeans Activation Framework (JAF) is included because JavaMail uses it. JAF provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate JavaBeans component to perform those operations.

Java API for XML Processing

The Java API for XML Processing (JAXP) supports the processing of XML documents using Document Object Model (DOM), Simple API for XML (SAX), and Extensible Stylesheet Language Transformations (XSLT). JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation.

JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts. Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the W3C schema..

Java API for XML-Based RPC

The Java API for XML-based RPC (JAX-RPC) uses the SOAP standard and HTTP, so client programs can make XML-based remote procedure calls (RPCs) over the Internet. JAX-RPC also supports WSDL, so you can import and export WSDL documents. With JAX-RPC and a WSDL, you can easily interoperate with clients and services running on Java-based or non-Java-based platforms such as .NET. For example, based on the WSDL document, a Visual Basic .NET client can be configured to use a web service implemented in Java technology, or a web service can be configured to recognize a Visual Basic .NET client.

JAX-RPC relies on the HTTP transport protocol. Taking that a step further, JAX-RPC lets you create service applications that combine HTTP with a Java technology version of the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols to establish basic or mutual authentication. SSL and TLS ensure message integrity by providing data encryption with client and server authentication capabilities.

Authentication is a measured way to verify whether a party is eligible and able to access certain information as a way to protect against the fraudulent use of a system or the fraudulent transmission of information. Information transported across the Internet is especially vulnerable to being intercepted and misused, so it's very important to configure a JAX-RPC web service to protect data in transit.

SOAP with Attachments API for Java

The SOAP with Attachments API for Java (SAAJ) is a low-level API on which JAX-RPC depends. SAAJ enables the production and consumption of messages that conform to the SOAP 1.1 specification and SOAP with Attachments note. Most developers do not use the SAAJ API, instead using the higher-level JAX-RPC API.

Java API for XML Registries

The Java API for XML Registries (JAXR) lets you access business and general-purpose registries over the web. JAXR supports the ebXML Registry and Repository standards and the emerging UDDI specifications. By using JAXR, developers can learn a single API and gain access to both of these important registry technologies.

Additionally, businesses can submit material to be shared and search for material that others have submitted. Standards groups have developed schemas for particular kinds of XML documents; two businesses might, for example, agree to use the schema for their industry's standard purchase order form. Because the schema is stored in a standard business registry, both parties can use JAXR to access it.

J2EE Connector Architecture

The J2EE Connector architecture is used by J2EE tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged in to any J2EE product. A *resource adapter* is a software component that allows J2EE application components to access and interact with the underlying resource manager of the EIS. Because a resource adapter is specific to its resource manager, typically there is a different resource adapter for each type of database or enterprise information system.

The J2EE Connector architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of J2EE-based web services with existing EISs that can be either synchronous or asynchronous. Existing applications and EISs integrated through the J2EE Connector architecture into the J2EE platform can be exposed as XML-based web services by using JAX-RPC and J2EE component models. Thus JAX-RPC and the J2EE Connector architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.

JDBC API

The JDBC API lets you invoke SQL commands from Java programming language methods. You use the JDBC API in an enterprise bean when you override the default container-managed persistence or have a session bean access the database. With container-managed persistence, database access operations are handled by the container, and your enterprise bean implementation contains no JDBC code or SQL commands. You can also use the JDBC API from a servlet or a JSP page to access the database directly without going through an enterprise bean.

The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the J2EE platform.

Java Naming and Directory Interface

The Java Naming and Directory Interface™ (JNDI) provides naming and directory functionality. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using JNDI, a J2EE application can store and retrieve any type of named Java object.

J2EE naming services provide application clients, enterprise beans, and web components with access to a JNDI naming environment. A *naming environment* allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI *naming context*.

A J2EE component locates its environment naming context using JNDI interfaces. A component creates a javax.naming.InitialContext object and looks up the environment naming context in InitialContext under the name java:comp/env. A component's naming environment is stored directly in the environment naming context or in any of its direct or indirect subcontexts.

A J2EE component can access named system-provided and user-defined objects. The names of system-provided objects, such as JTA UserTransaction objects, are stored in the environment naming context, java:comp/env. The J2EE platform allows a component to name user-defined objects, such as enterprise beans, environment entries, JDBC DataSource objects, and message connections. An object should be named within a subcontext of the naming environment according to the type of the object. For example, enterprise beans are named within the subcontext java:comp/env/ejb, and JDBC DataSource references in the subcontext java:comp/env/jdbc.

Because JNDI is independent of any specific implementation, applications can use JNDI to access multiple naming and directory services, including existing naming and directory services such as LDAP, NDS, DNS, and NIS. This allows J2EE applications to coexist with legacy applications and systems.

Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) provides a way for a J2EE application to authenticate and authorize a specific user or group of users to run it.

JAAS is a Java programing language version of the standard Pluggable Authentication Module (PAM) framework, which extends the Java 2 Platform security architecture to support user-based authorization.

Simplified Systems Integration

The J2EE platform is a platform-independent, full systems integration solution that creates an open marketplace in which every vendor can sell to every customer. Such a marketplace encourages vendors to compete, not by trying to lock customers into their technologies but instead by trying to outdo each other in providing products and services that benefit customers, such as better performance, better tools, or better customer support.

The J2EE APIs enable systems and applications integration through the following:

- Unified application model across tiers with enterprise beans
- Simplified request-and-response mechanism with JSP pages and servlets
- Reliable security model with JAAS
- XML-based data interchange integration with JAXP, SAAJ, and JAX-RPC
- Simplified interoperability with the J2EE Connector architecture
- Easy database connectivity with the JDBC API
- Enterprise application integration with message-driven beans and JMS, JTA, and JNDI

Sun Java System Application Server Platform Edition 8

The Sun Java System Application Server Platform Edition 8 is a fully compliant implementation of the J2EE 1.4 platform. In addition to supporting all the APIs described in the previous sections, the Application Server includes a number of J2EE technologies

and tools that are not part of the J2EE 1.4 platform but are provided as a convenience to the developer.

This section briefly summarizes the technologies and tools that make up the Application Server, and instructions for starting and stopping the Application Server, starting the Admin Console, starting deploytool, and starting and stopping the Derby database server. Other chapters explain how to use the remaining tools.

Technologies

The Application Server includes two user interface technologies--JavaServer Pages Standard Tag Library and JavaServer™ Faces--that are built on and used in conjunction with the J2EE 1.4 platform technologies Java servlet and JavaServer Pages.

JavaServer Pages Standard Tag Library

The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in your JSP applications, you employ a single, standard set of tags. This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of the tags is optimized.

JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

JavaServer Faces

JavaServer Faces technology is a user interface framework for building web applications. The main components of JavaServer Faces technology are as follows:

- A GUI component framework.
- A flexible model for rendering components in different kinds of HTML or different markup languages and technologies. A Renderer object generates

the markup to render the component and converts the data stored in a model object to types that can be represented in a view.

- A standard RenderKit for generating HTML/4.01 markup.

The following features support the GUI components:

- Input validation
- Event handling
- Data conversion between model objects and components
- Managed model object creation
- Page navigation configuration

All this functionality is available via standard Java APIs and XML-based configuration files.

Tools

The Application Server contains the tools listed in Table 1-1. Basic usage information for many of the tools appears throughout the tutorial.

Table 1-1 Application Server Tools

Component	Description
Admin Console	A web-based GUI Application Server administration utility. Used to stop the Application Server and manage users, resources, and applications.
asadmin	A command-line Application Server administration utility. Used to start and stop the Application Server and manage users, resources, and applications.
asant	A portable command-line build tool that is an extension of the Ant tool developed by the Apache Software Foundation. asant contains additional tasks that interact with the Application Server administration utility.
appclient	A command-line tool that launches the application client container and invokes the client application packaged in the application client JAR file.
capture-schema	A command-line tool to extract schema information from a database, producing a schema file that the Application Server can use for container-managed persistence.

deploytool	A GUI tool to package applications, generate deployment descriptors, and deploy applications on the Application Server.
package-appclient	A command-line tool to package the application client container libraries and JAR files.
Derby database	A copy of the open source Derby database server.
verifier	A command-line tool to validate J2EE deployment descriptors.
wscompile	A command-line tool to generate stubs, ties, serializers, and WSDL files used in JAX-RPC clients and services.

Web Applications

A web application is a dynamic extension of a web or application server. There are two types of web applications:

- *Presentation-oriented*: A presentation-oriented web application generates interactive web pages containing various types of markup language (HTML, XML, and so on) and dynamic content in response to requests.
- *Service-oriented*: A service-oriented web application implements the endpoint of a web service. Presentation-oriented applications are often clients of service-oriented web applications

In the Java 2 platform, *web components* provide the dynamic extension capabilities for a web server. Web components are either Java servlets, JSP pages, or web service endpoints. The interaction between a web client and a web application is illustrated in Figure 3-1. The client sends an HTTP request to the web server. A web server that implements Java Servlet and JavaServer Pages technology converts the request into an HTTPServletRequest object. This object is delivered to a web component, which can interact with JavaBeans components or a database to generate dynamic content. The web component can then generate an HTTPServletResponse or it can pass the request to another web component. Eventually a web component generates a HTTPServletResponse object. The web server converts this object to an HTTP response and returns it to the client.

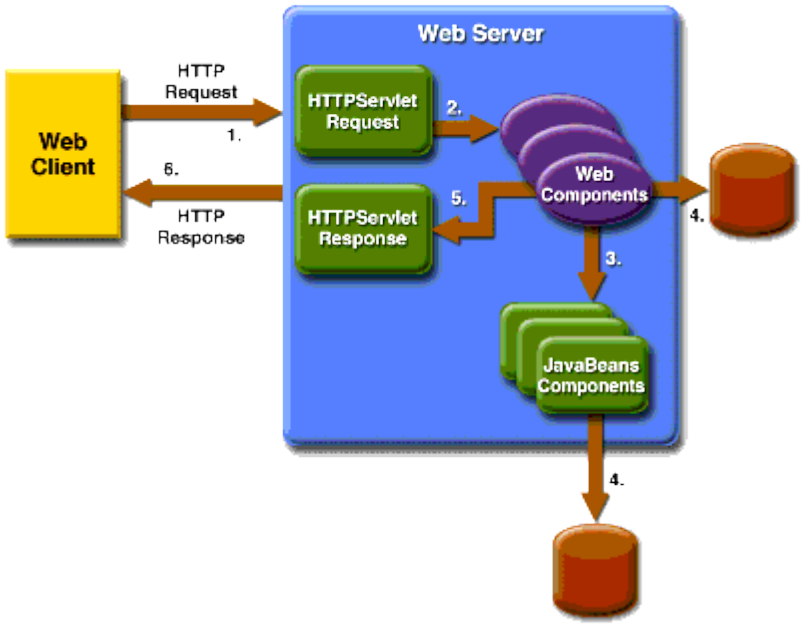


Figure 3-1 Java Web Application Request Handling

Servlets are Java programming language classes that dynamically process requests and construct responses. *JSP pages* are text-based documents that execute as servlets but allow a more natural approach to creating static content. Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited for service-oriented applications (web service endpoints are implemented as servlets) and the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data. JSP pages are more appropriate for generating text-based markup such as HTML, Scalable Vector Graphics (SVG), Wireless Markup Language (WML), and XML.

Since the introduction of Java Servlet and JSP technology, additional Java technologies and frameworks for building interactive web applications have been developed. These technologies and their relationships are illustrated in Figure 3-2.

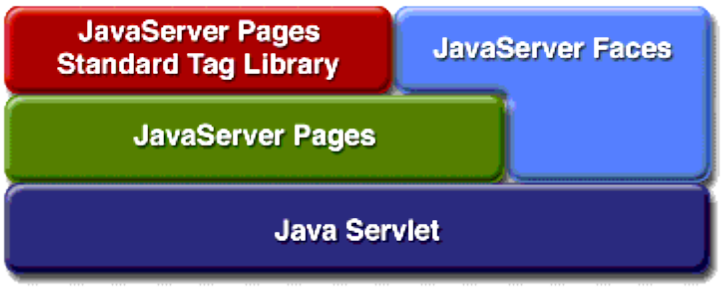


Figure 3-2 Java Web Application Technologies

Notice that Java Servlet technology is the foundation of all the web application technologies. Each technology adds a level of abstraction that makes web application prototyping and development faster and the web applications themselves more maintainable, scalable, and robust.

Web components are supported by the services of a runtime platform called a *web container*. A web container provides services such as request dispatching, security, concurrency, and life-cycle management. It also gives web components access to APIs such as naming, transactions, and email.

Certain aspects of web application behavior can be configured when the application is installed, or *deployed*, to the web container. The configuration information is maintained in a text file in XML format called a *web application deployment descriptor* (DD). A DD must conform to the schema described in the Java Servlet Specification.

Most web applications use the HTTP protocol, and support for HTTP is a major aspect of web components.

This chapter gives a brief overview of the activities involved in developing web applications. First we summarize the web application life cycle. Then we describe how to package and deploy very simple web applications on the Application Server. We move on to configuring web applications and discuss how to specify the most commonly used configuration parameters. We then introduce an example--Duke's Bookstore--that we use to illustrate all the J2EE web-tier technologies and we describe how to set up the shared

components of this example. Finally we discuss how to access databases from web applications a

Web Application Life Cycle

A web application consists of web components, static resource files such as images, and helper classes and libraries. The web container provides many supporting services that enhance the capabilities of web components and make them easier to develop. However, because a web application must take these services into account, the process for creating and running a web application is different from that of traditional stand-alone Java classes. The process for creating, deploying, and executing a web application can be summarized as follows:

1. Develop the web component code.
2. Develop the web application deployment descriptor.
3. Compile the web application components and helper classes referenced by the components.
4. Optionally package the application into a deployable unit.
5. Deploy the application into a web container.

Access a URL that references the web application. Configuring Web Applications

Web applications are configured via elements contained in the web application deployment descriptor. The deploytool utility generates the descriptor when you create a WAR and adds elements when you create web components and associated classes. You can modify the elements via the inspectors associated with the WAR.

The following sections give a brief introduction to the web application features you will usually want to configure.

In the following sections, examples demonstrate procedures for configuring the Hello, World application. If Hello, World does not use a specific configuration feature, the section gives references to other examples that illustrate how to specify the deployment

descriptor element and describes generic procedures for specifying the feature using deploytool. Extended examples that demonstrate how to use deploytool appear in later tutorial chapters.

Servlets

This first chapter answers the question "What is a Servlet?", shows typical uses for Servlets, compares Servlets to CGI programs and explains the basics of the Servlet architecture and the Servlet lifecycle. It also gives a quick introduction to HTTP and its implementation in the HttpServlet class.

Servlets are modules of Java code that run in a server application (hence the name "Servlets", similar to "Applets" on the client side) to answer client requests. Servlets are not tied to a specific client-server protocol but they are most commonly used with HTTP and the word "Servlet" is often used in the meaning of "HTTP Servlet".

Servlets make use of the Java standard extension classes in the packages javax.servlet (the basic Servlet framework) and javax.servlet.http (extensions of the Servlet framework for Servlets that answer HTTP requests). Since Servlets are written in the highly portable Java language and follow a standard framework, they provide a means to create sophisticated server extensions in a server and operating system independent way.

Typical uses for HTTP Servlets include:

- Processing and/or storing data submitted by an HTML form.
- Providing dynamic content, e.g. returning the results of a database query to the client.
- Managing state information on top of the stateless HTTP, e.g. for an online shopping cart system which manages shopping carts for many concurrent customers and maps every request to the right customer.

Servlets vs. CGI

The traditional way of adding functionality to a Web Server is the Common Gateway Interface (CGI), a language-independent interface that allows a server to start an external process which gets information about a request through environment variables, the command line and its standard input stream and writes response data to its standard output stream. Each request is answered in a separate process by a separate instance of the CGI program, or CGI script (as it is often called because CGI programs are usually written in interpreted languages like Perl).

Servlets have several advantages over CGI:

- A Servlet does not run in a separate process. This removes the overhead of creating a new process for each request.
- A Servlet stays in memory between requests. A CGI program (and probably also an extensive runtime system or interpreter) needs to be loaded and started for each CGI request.
- There is only a single instance which answers all requests concurrently. This saves memory and allows a Servlet to easily manage persistent data.
- A Servlet can be run by a Servlet Engine in a restrictive Sandbox (just like an Applet runs in a Web Browser's Sandbox) which allows secure use of untrusted and potentially harmful Servlets.

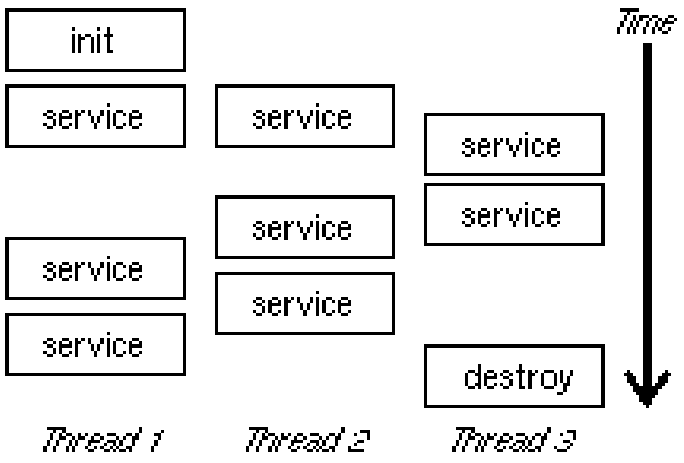
The Basic Servlet Architecture

A Servlet, in its most general form, is an instance of a class which implements the javax.servlet.Servlet interface. Most Servlets, however, extend one of the standard implementations of that interface, namely javax.servlet.GenericServlet and javax.servlet.http.HttpServlet. In this tutorial we'll be discussing only HTTP Servlets which extend the javax.servlet.http.HttpServlet class.

In order to initialize a Servlet, a server application loads the Servlet class (and probably other classes which are referenced by the Servlet) and creates an instance by calling the no-args constructor. Then it calls the Servlet's init(ServletConfig config) method. The Servlet should performe one-time setup procedures in this method and store the ServletConfig object so that it can be retrieved later by calling the Servlet's getServletConfig() method. This is handled by GenericServlet. Servlets which extend GenericServlet (or its subclass HttpServlet) should call super.init(config) at the beginning of the init method to make use of this feature. The ServletConfig object contains Servlet parameters and a reference to the Servlet's ServletContext. The init method is guaranteed to be called only once during the Servlet's lifecycle. It does not need to be thread-safe because the service method will not be called until the call to init returns.

When the Servlet is initialized, its service(ServletRequest req, ServletResponse res) method is called for every request to the Servlet. The method is called concurrently (i.e. multiple threads may call this method at the same time) so it should be implemented in a thread-safe manner.

When the Servlet needs to be unloaded (e.g. because a new version should be loaded or the server is shutting down) the destroy() method is called. There may still be threads that execute the service method when destroy is called, so destroy has to be thread-safe. All resources which were allocated in init should be released in destroy. This method is guaranteed to be called only once during the Servlet's lifecycle.



HTTP

Before we can start writing the first Servlet, we need to know some basics of HTTP ("HyperText Transfer Protocol"), the protocol which is used by a WWW client (e.g. a browser) to send a request to a Web Server.

HTTP is a request-response oriented protocol. An HTTP request consists of a request method, a URI, header fields and a body (which can be empty). An HTTP response contains a result code and again header fields and a body.

The service method of HttpServlet dispatches a request to different Java methods for different HTTP request methods. It recognizes the standard HTTP/1.1 methods and should not be overridden in subclasses unless you need to implement additional methods. The recognized methods are GET, HEAD, PUT, POST, DELETE, OPTIONS and TRACE. Other methods are answered with a *Bad Request* HTTP error. An HTTP method *XXX* is dispatched to a Java method do*Xxx*, e.g. *GET* -> do*Get*. All these methods expect the parameters "(HttpServletRequest req, HttpServletResponse res)". The methods doOptions and doTrace have suitable default implementations and are usually not overridden. The HEAD method (which is supposed to return the same header lines that a GET method would return, but doesn't include a body) is performed by calling doGet and ignoring any output that is written by this method. That leaves us with the methods doGet, doPut, doPost and doDelete whose default implementations in HttpServlet return a *Bad Request* HTTP error. A subclass of HttpServlet overrides one or more of these methods to provide a meaningful implementation.

The request data is passed to all methods through the first argument of type HttpServletRequest (which is a subclass of the more general ServletRequest class). The response can be created with methods of the second argument of type HttpServletResponse (a subclass of ServletResponse).

When you request a URL in a Web Browser, the GET method is used for the request. A GET request does not have a body (i.e. the body is empty). The response should contain a body with the response data and header fields which describe the body (especially

Content-Type and Content-Encoding). When you send an HTML form, either GET or POST can be used. With a GET request the parameters are encoded in the URL, with a POST request they are transmited in the body. HTML editors and upload tools use PUT requests to upload resources to a Web Server and DELETE requests to delete resources.

The Servlet Environment

This chapter shows how to access resources of the Web Server and communicate with Servlets and other kinds of active resources (e.g. JSP documents, CGI programs). The examples in this chapter are often only fragments of Java source code and not necessarily complete Servlets.

Inter Servlet Communication

This section shows how to

- call a method of another Servlet

Servlets are not alone in a Web Server. They have access to other Servlets in the same *Servlet Context* (usually a Servlet directory), represented by an instance of javax.servlet.ServletContext. The ServletContext is available through the ServletConfig object's getServletContext method.

A Servlet can get a list of all other Servlets in the Servlet Context by calling getServletNames on the ServletContext object. A Servlet for a known name (probably obtained through getServletNames) is returned by getServlet. Note that this method can throw a ServletException because it may need to load and initialize the requested Servlet if this was not already done.

After obtaining the reference to another Servlet that Servlet's methods can be called. Methods which are not declared in javax.servlet.Servlet but in a subclass thereof can be called by casting the returned object to the required class type.

Note that in Java the identity of a class is not only defined by the class name but also by the ClassLoader by which it was loaded. Web servers usually load each Servlet with a different class loader. This is necessary to reload Servlets on the fly because single classes cannot be replaced in the running JVM. Only a ClassLoader object with all loaded classes can be replaced.

This means that classes which are loaded by a Servlet class loader cannot be used for inter-Servlet communication. A class literal FooServlet (as used in a type cast like "FooServlet foo = (FooServlet)context.getServlet("FooServlet")") which is used in class BarServlet is different from the class literal FooServlet as used in FooServlet itself.

A way to overcome this problem is using a superclass or an interface which is loaded by the system loader and thus shared by all Servlets. In a Web Server which is written in Java those classes are usually located in the class path (as defined by the CLASSPATH environment variable).

Communication with Active Server Resources

This section shows how to

- call another Servlet (or any other kind of active resource) to process a request

A Servlet can make a request to an active resource on the web server just like a client can (by requesting a URL). The Servlet API supports a more direct way of accessing server resources from within a Servlet which is running in the server than opening a socket connection back to the server. A Servlet can either hand off a request to a different resource or include the response which is created by that resource in its own response. It is also possible to supply user-defined data when calling an active resource which provides for an elegant way of doing inter-Servlet communication.

In the doGet method we first check for the existence of the item argument. If it is present (second branch of the if statement) the Servlet is responding with an HTML document in

the usual way. In the middle of the document the response body of the ItemServlet is included by asking the ItemServlet's RequestDispatcher (whis is obtained through the ServletContext) to perform that operation.

If no item attribute was specified, the request is delegated to the ErrorServlet in a similar way. Note that this time we are using the RequestDispatcher's forward method (instead of include). This method can be called only once and only if neither getOutputStream nor getWriter has been called, but it allows the included Servlet to set headers and the response code (which is required to create a proper HTTP error response message).

The ItemServlet gets its item argument from the query string. That way it can also be accessed directly via an HTTP request, but argument values have to be represented as url-encoded strings (which is no problem in this case). The ErrorServlet takes an exception argument of type java.lang.Exception which is provided as a request attribute. The ErrorServlet uses the ServletRequest method getAttribute to read the attribute.

Note that a server which supports load balancing could run the Servlets on different JVMs. All custom request attributes should be *serializable* to allow them to be moved from one JVM to another.

Accessing Passive Server Resources

This section shows how to

- access a resource in the server's document tree

Passive server resources (e.g. static HTML pages which are stored in local files) are not accessed with RequestDispatcher objects. The ServletContext method getResource(String path) returns a URL object for a resource specified by a local URI (e.g. "/" for the server's document root) which can be used to examine the resource.

If you only want to read the resource's body you can directly ask the ServletContext for an InputStream with the getResourceAsStream(String path) method.

Accessing Servlet Resources

This section shows how to

- access resources which belong to a Servlet

A Servlet may need to access additional resources like configuration files whose locations should not need to be specified in init parameters. Those resources can be accessed with the methods `getResource(String name)` and `getResourceAsStream(String name)` of the `java.lang.Class` object which represents the Servlet's class.

Example. The following code gets an `InputStream` for a configuration file named `myservlet.cfg` which resides in the same directory as the class in which the code is executed:

```
InputStream confIn =

  getClass().getResourceAsStream("myservlet.cfg");
```

Note that the Servlet engine's Servlet class loader must implement the `getResource` and `getResourceAsStream` methods in order for this to work. *This may not be the case with all Servlet engines.*

Sharing Servlet Resources

This section shows how to

- share data between Servlets

Version 2.1 of the Servlet API offers a new way of sharing named objects between all the Servlets in a Servlet context (and also other contexts, as you'll see below) by binding the objects to the `ServletContext` object which is shared by several Servlets.

The ServletContext class has several methods for accessing the shared objects:

- public void setAttribute(String name, Object object) adds a new object or replaces an old object by the specified name. The attribute name should follow the same naming convention as a package name (e.g. a Servlet com.foo.fooservlet.FooServlet could have an attribute com.foo.fooservlet.bar).

Just like a custom ServletRequest attribute, an object which is stored as a ServletContext attribute should also be *serializable* to allow attributes to be shared by Servlets which are running in different JVMs on different machines in a load-balancing server environment.

- public Object getAttribute(String name) returns the named object or null if the attribute does not exist.

In addition to the user-defined attributes there may also be predefined attributes which are specific to the Servlet engine and provide additional information about a Servlet(Context)'s environment.

- public Enumeration getAttributeNames() returns an Enumeration of the names of all available attributes.
- public void removeAttribute(String name) removes the attribute with the specified name if it exists.

The separation of Servlets into Servlet contexts depends on the Servlet engine. The ServletContext object of a Servlet with a known local URI can be retrieved with the method public ServletContext getContext(String uripath) of the Servlet's own ServletContext. This method returns null if there is no Servlet for the specified path or if this Servlet is not allowed to get the ServletContext for the specified path due to security restrictions.

JAVA Server Pages™

The Java Server Pages™ (JSP) technology provides a simplified, fast way to create web pages that display dynamically-generated content. JSP technology was designed to make it easier and faster to build web-based applications that work with a wide variety of web servers, application servers, browsers and development tools.

This paper provides an overview of the JSP technology, describing the background in which it was developed and the overall goals for the technology. It also describes the key components of a Java™ technology-based page, in the context of a simple example.

Developing Web-based Applications: A Background.

In its short history, the Worldwide Web has evolved from a network of basically static information displays to a mechanism for trading stocks and buying books. There seems to be almost no limit to the possible uses for web-based clients in diverse applications.

Applications that can make use of browser-based clients have several advantages over traditional client/server based applications. These include nearly unlimited client access and greatly simplified application deployment and management. (To update an application, a developer only needs to change one server-based program, not thousands of client-installed applications.) As a result, the software industry is moving quickly toward building multi-tiered applications using browser-based clients.

These increasingly sophisticated web-based applications require changes in development technology. Static HTML is fine for displaying relatively static content; the challenge has been creating interactive web-based applications, in which the content of the page is based on a user request or system status, not pre-defined text.

An early solution to this problem was the CGI-BIN interface; developers wrote individual programs to this interface, and web-based applications called the programs through the web server. This solution has significant scalability problems -- each new CGI request launches a new process on the server. If multiple users access the program concurrently,

these processes consume all of the web server's available resources and the performance slows to a grind.

Individual web server vendors have tried to simplify web application development providing "plug-ins" and APIs for their servers. These solutions are web-server specific, and don't address the problem across multiple vendor solutions. For example, Microsoft's Active Server Pages™ (ASP) technology makes it easier to create dynamic content on a web page, but only works with Microsoft IIS or Personal Web Server.

Other solutions exist, but they are not necessarily easy for the average page designer to deploy. Technologies such as Java Servlets, for example, make it easier to write server-based code using the Java programming language for interactive applications. A Java Servlet is a Java technology-based program that runs on the server (as opposed to an applet, which runs on the browser). Developers can write Servlets that take an HTTP request from the web browser, generate the response dynamically (possibly querying databases to fulfill the request) and then send a response containing an HTML or XML document to the browser.

Using this approach, the entire page must be composed in the Java Servlet. If a developer or web master wanted to tune the appearance of the page, they would have to edit and recompile the Java Servlet, even if the logic were already working. With this approach, generating pages with dynamic content still requires application development expertise.

What is needed, clearly, is an industry-wide solution for creating pages with dynamically-generated content. This solution should address the limitations of current alternatives by: Working on any web or application server Separating the application logic from the appearance of the page Allowing fast development and testing Simplifying the process of developing interactive web-based applications.

The JavaServer Pages (JSP) technology was designed to fit this need. The JSP specification is the result of extensive industry cooperation between vendors of web servers, application servers, transactional systems, and development tools. Sun Microsystems developed the specification to integrate with and leverage existing

expertise and tools support for the Java programming environment, such as Java Servlets and JavaBeans™. The result is a new approach to developing web-based applications that extends powerful capabilities to page designers using component-based application logic.

The JavaServer Pages Technology Approach to Web Application Development

In developing the JSP specification, Sun Microsystems worked with a number of leading web server, application server and development tool vendors, as well as a diverse and experienced development community. The result is an approach that balances portability with ease-of-use for application and page developers.

JSP technology speeds the development of dynamic web pages in a number of ways:

Separating content generation from presentation

Using JSP technology, web page developers use HTML or XML tags to design and format the results page. They use JSP tags or scriptlets to generate the dynamic content (the content that changes according to the request, such as requested account information or the price of a specific bottle of wine). The logic that generates the content is encapsulated in tags and JavaBeans components and tied together in scriptlets, all of which are executed on the server side. If the core logic is encapsulated in tags and beans, then other individuals, such as web masters and page designers, can edit the JSP page without affecting the generation of the content.

On the server, a JSP engine interprets JSP tags and scriptlets, generates content (for example, by accessing JavaBeans components, accessing a database with JDBC™ technology, or including files), and sends the results back in the form of an HTML (or XML) page to the browser. This helps authors protect proprietary code while ensuring complete portability for any HTML-based web browser.

Emphasizing reusable components

Most JSP pages rely on reusable, cross-platform components (JavaBeans or Enterprise JavaBeans™ components) to perform the more complex processing required of the

application. Developers can share and exchange components that perform common operations, or make them available to larger user or customer communities. The component-based approach speeds overall development and lets organizations leverage their existing expertise and development efforts for optimal results.

Simplifying page development with tags

Web page developers are not always programmers familiar with scripting languages. The JavaServer Pages technology encapsulates much of the functionality required for dynamic content generation in easy-to-use, JSP-specific XML tags. Standard JSP tags can access and instantiate JavaBeans components, set or retrieve bean attributes, download applets, and perform other functions that are otherwise more difficult and time-consuming to code.

The JSP technology is extensible through the development of customized tag libraries. Over time, third-party developers and others will create their own tag libraries for common functions. This lets web page developers work with familiar tools and constructs, such as tags, to perform sophisticated functions.

The JSP technology integrates easily into a variety of application architectures, leveraging existing tools and skills, and scaling to support enterprise-wide distributed applications. As part of the Java technology-enabled family, and an integral part of the Java 2, Enterprise Edition architecture, the JSP technology can support highly complex web-based applications.

Because the native scripting language for JSP pages is based on the Java programming language, and because all JSP pages are compiled into Java Servlets, JSP pages have all of the benefits of Java technology, including robust memory management and security.

As part of the Java platform, JSP shares the Write Once, Run Anywhere™ characteristics of the Java programming language. As more vendors add JSP support to their products, you can use servers and tools of your choice, changing tools or servers without affecting current applications.

When integrated with the Java 2 Platform, Enterprise Edition (J2EE) and Enterprise JavaBeans technology, JSP pages will provide enterprise-class scalability and performance necessary for deploying web-based applications across the virtual enterprise.

What Does a JSP Page Look Like?

A JSP page looks like a standard HTML or XML page, with additional elements that the JSP engine processes and strips out. Typically, the JSP elements create text that is inserted into the results page.

The JSP technology is best described using an example. The following JSP page is very simple; it prints the day of the month and the year, and welcomes you with either "Good Morning" or "Good Afternoon," depending on the time of day.

The page combines ordinary HTML with a number of JSP elements: Calls to a clock JavaBean component An inclusion of an external file (for copyright information) JSP expressions and scriptlets

```
<HTML>
<%@ page language=="java" imports=="com.wombat.JSP.*" %>
<H1>Welcome</H1>
<P>Today is </P>
<jsp:useBean id=="clock" class=="calendar.jspCalendar" />
<UL>
<LI>Day: <%==clock.getDayOfMonth() %>
<LI>Year: <%==clock.getYear() %>
</UL>
<% if (Calendar.getInstance().get(Calendar.AM_PM) ==== Calendar.AM) { %>
Good Morning
<% } else { %>
Good Afternoon
<% } %>
```

```
<%@ include file=="copyright.html" %>
</HTML>
```

The page includes the following components:

A JSP directive passes information to the JSP engine. In this case, the first line indicates the location of some Java programming language extensions to be accessible from this page. Directives are enclosed in <%@ and %> markers.

Fixed template data: Any tags that the JSP engine does not recognize it passes on with the results page. Typically, these will be HTML or XML tags. This includes the Unordered List and H1 tags in the example above.

JSP actions, or tags: These are typically implemented as standard tags or customized tags, and have an XML tag syntax. In the example, the jsp:useBean tag instantiates the Clock JavaBean on the server.

An expression: The JSP engine evaluates anything between <%== and %> markers. In the List Items above, the values of the Day and Year attributes of the Clock bean are returned as a string and inserted as output in the JSP file. In the example above, the first list item will be the day of the year, and the second item the year.

A scriptlet is a small script that performs functions not supported by tags or ties everything together. The native scripting language for JSP 1.0 software is based on the Java programming language. The scriptlet in the above sample determines whether it is AM or PM and greets the user accordingly (for daytime users, at any rate).

The example may be trivial, but the technology is not. Businesses can encapsulate critical processing in server-side Beans, and web developers can easily access that information, using familiar syntax and tools. Java-based scriptlets provide a flexible way to perform other functions, without requiring extensive scripting. The page as a whole is legible and comprehensible, making it easier to spot or prevent problems and to share work.

A few of these components are described in more detail below.

JSP Directives

JSP pages use JSP directives to pass instructions to the JSP engine. These may include the following:

JSP *page directives* communicate page-specific information, such as buffer and thread information or error handling.

Language directives specify the scripting language, along with any extensions.

The *include directive* (shown in the example above) can be used to include an external document in the page. A good example is a copyright file or company information, file -- it is easier to maintain this file in one central location and include it in several pages than to update it in each JSP page. However, the included file can also be another JSP file.

A *taglib directive* indicates a library of custom tags that the page can invoke.

JSP Tags

Most JSP processing will be implemented through JSP-specific XML-based tags. JSP 1.0 includes a number of standard tags, referred to as the core tags. These include:

jsp:useBean: This tag declares the usage of an instance of a JavaBeans component. If the Bean does not already exist, then the JavaBean component instantiates and registers the tag.

jsp:setProperty: This sets the value of a property in a Bean.

jsp:getProperty: This tag gets the value of a Bean instance property, converts it to a string, and puts It in the implicit object "out".

jsp:include:

jsp:forward:

The 1.1 release will include additional standard tags.

The advantage of tags is that they are easy to use and share between applications. The real power of a tag-based syntax comes with the development of custom tag libraries, in which tool vendors or others can create and distribute tags for specific purposes.

Scripting Elements

JSP pages can includes include small scripts, called scriptlets, in a page. A scriplet is a code fragment, executed at request time processing. Scriptlets may be combined with static elements on the page (as in the example above) to create a dynamically-generated page.

Scripts are delineated within <% and %> markers. Anything within those markers will be evaluated by the scripting language engine, in our example the Java virtual machine on the host.

The JSP specification supports all of the usual script elements, including expressions and declarations.

Application Models for JSP Pages

A JSP page is executed by a JSP engine, which is installed in a web server or a JSP-enabled application server. The JSP engine receives requests from a client to a JSP page, and generates responses from the JSP page to the client.

JSP pages are typically compiled into Java Servlets. Java Servlets are a standard Java extension, described in more detail at www.java.sun.com. The page developer has access to the complete Java application environment, with all of the scalability and portability of the Java technology-enabled family.

When a JSP page is first called, if it does not yet exist, it is compiled into a Java Servlet class and stored in the server memory. This enables very fast responses for subsequent calls to that page. (This avoids the CGI-bin problem of spawning a new processes for each HTTP request, or the runtime parsing required by server-side includes.)

JSP pages may be included in a number of different application architectures or models. JSP pages may be used in combination with different protocols, components and formats. The following sections describe a few of the possibilities.

A Simple Application

In a simple implementation, the browser directly invokes a JSP page, which itself generates the requested content (perhaps invoking JDBC to get information directly from a database). The JSP page can call JDBC or Java Blend™ components to generate results, and creates standard HTML that it sends back to the browser as a result.

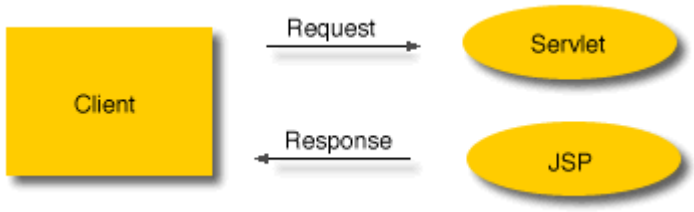
This model basically replaces the CGI-BIN concept with a JSP page (compiled as a Java Servlet). This method has the following advantages:

It is simple and fast to program The page author can easily generate dynamic content based on the request and state of the resources.

This architecture works well for many applications, but it does not scale for a large number of simultaneous Web-based clients accessing scarce enterprise resources, since each must establish or share a connection to the content resource in question. For example, if the JSP page accesses a database, it may generate many connections to the database, which can affect the database performance.

A Flexible Application with Java Servlets

In another possible configuration, the Web-based client may make a request directly to a Java Servlet, which actually generates the dynamic content, wraps the results into a result bean and invokes the JSP page. The JSP page accesses the dynamic content from the bean and sends the results (as HTML) to the browser.



This approach creates more reusable components that can be shared between applications, and may be implemented as part of a larger application. It still has scalability issues in terms of handling connections to enterprise resources, such as databases.

Scalable Processing with Enterprise JavaBeans Technology

The JSP page can also act as a middle tier within an Enterprise JavaBeans (EJB) architecture. In this case, the JSP page interacts with back end resources via an Enterprise JavaBeans component.

The EJB component manages access to the back end resources, which provides scalable performance for high numbers of concurrent users. For e-commerce or other applications, the EJB manages transactions and underlying security. This simplifies the JSP page itself. This model will be supported by the Java 2 Enterprise Edition (J2EE) platform.

Integrating XML Technology in JSP Pages

JSP pages can be used to generate both XML and HTML pages.

For simple XML generation, developers can include XML tags and static template portions of the JSP page. For dynamic XML generation, use server-based Beans and customized tags that generate XML output.

JSP pages are not incompatible with XML tools. Although Sun designed the JSP specification so that JSP pages would be easy to author, even by hand, the JSP specification also provides a mechanism for creating an XML version of any JSP page. In this way, XML tools can author and manipulate JSP pages.

You can use JSP pages with XML-based tools by converting JSP tags and elements to their XML-compatible equivalents. For example, a scriptlet can be included within <% and %>, or within the XML-based tags <jsp:scriptlet> and </jsp:scriptlet>. In fact, it is possible to convert a JSP page into an XML page by following a few simple steps, including:

adding a JSP root element converting elements and directives into XML-compatible equivalents creating CDATA elements for all other (typically non-JSP) elements on the page

With this XML-compatible alternative approach, page designers creating HTML pages still have an easy-to-use environment for quickly creating dynamic web pages, while XML-based tools and services can integrate JSP pages and work with JSP-compliant servers.

The Future for JSP Technology

JSP technology is designed to be an open, extensible standard for building dynamic web pages. Developers will use JSP pages to create portable web applications that can run

with different web and application servers for different markets, using whatever authoring tools fit their market and their needs.

By working with a consortium of industry leaders, Sun has ensured that the JSP specification is open and portable. You should be able to author JSP pages anywhere and deploy them anywhere, using any client and server platforms. Over time, tool vendors and others will extend the functionality of the platform by providing customized tag libraries for specialized functions.

4.3 Packages

Packages are containers for classes that are used to keep the class name space compartmentalized. The package is both a naming and visibility control mechanism. We can define classes inside a package that are not accessible by code outside the package.

Java Media Framework (JMF)

The Java Media Framework API (JMF) enables audio, video and other time-based media to be added to applications and applets built on Java technology. This optional package, which can capture, playback, stream, and transcode multiple media formats, extends the Java 2 Platform, Standard Edition (J2SE) for multimedia developers by providing a powerful toolkit to develop scalable, cross-platform technology.

JMF 1.0

The JMF API is being developed in stages. JMF 1.0, also known as the "Java Media Player", is the first release of the JMF API and concentrates on media "playback", that is the synchronization, control, processing, and presentation of compressed streaming and stored time-based media, such as audio, video and MIDI. Sun released two reference implementations of JMF 1.0 for Windows and Solaris.

JMF 1.1

This is a dot release which added two version of JMF written entirely in the Java programming language, allowing JMF to be used on all JDK 1.1 Java Compatible systems. Note this is not a new API release; the same JMF 1.0 API was used for both JMF 1.0 and 1.1 implementations from Sun Microsystems, Inc.

JMF 2.0

JMF 2.0 is the second release of the JMF API. Where JMF 1.0 and 1.1 provide a versatile player for time-based media, JMF 2.0 features media capture, streaming, transcoding, a pluggable codec architecture, and greater control over media data so that developers can make greater customizations and optimizations.

Sun and IBM worked together to define the API and release three sample implementations of JMF 2.0: one written entirely in the Java programming language, and optimized versions for Solaris/SPARC and Windows.

JMF 2.1

This is a dot release which added SunRay and Linux support, increased support with open standard RTP-based video servers such as Apple's Quicktime Streaming Server and Sun's MCSS, and many bug fixes and optimizations.

Note this is not a new API release; the same JMF 2.0 API was used for both JMF 2.0 and 2.1 implementations from Sun Microsystems, Inc.

JMF 2.1.1

This is dot dot release contains new features and optimizations, including:

- Improved RTP API

- Support for H.263/1998 (RFC 2429) - now can interoperate with Darwin based RTSP servers.
- Direct Audio Renderer and Capturer
- Performance enhancements with new Java compilers
- JMF Customizer added to all JMF versions
- HTTPS, JAWT, UNC pathnames support
- Numerous bug fixes and optimizations

JMF 2.1.1 supports a wide array of media types, including

- **protocols:** FILE, HTTP, FTP, RTP
- **audio:** AIFF, AU, AVI, GSM, MIDI, MP2, MP3*, QT, RMF, WAV
- **video:** AVI, MPEG-1, QT, H.261, H.263
- **other:** Hot Media

*MP3 is supported only on the Windows platform.

JMF 2.1.1 will run on Windows 95/98/NT 4.0/2000, Solaris/SPARC, and any Java Compatible platforms. The pure Java version of JMF 2.1.1 will now run on Apple's MRJ 2.1.4; however there are a/v synchronization problems since Apple added a 6 second audio buffer to that MRJ.

RTP/RTSP streaming servers work with JMF 2.1.1

JMF 2.1.1 has been tested and is interoperable with the following tools and applications:

- Apple QuickTime Streaming Server - Apple Computer, Inc.
  Commercial RTP server for the Mac platform

- Darwin Streaming Server - Apple Computer, Inc.
  Public Source RTP server for the Linux, Solaris and other platforms

- ShowMe TV - Sun Microsystems
  Commercial RTP server/client for Solaris/SPARC

- Media Central Streaming Server - Sun Microsystems
  High capicity, commercial RTP server for Solaris/SPARC

- vat/vic - Lawrence Berkeley Labs
  Publically available audio and video RTP tools
  Runs on most Unix systems & Windows

JMF Registry

JMF 2.1.1 maintains a registry of available plug-in, package prefixes and other settings in a file called jmf properties. This is a binary file and should only be modified using the provided JMFRegistry application. This application is a part of the JMF 2.1.1 jmf.jar file and can be run as "java JMFRegistry". It requires that you have Swing-1.1 in your CLASSPATH (or you can use JDK 1.2 or later).

JMF 2.1.1 RTP/RTSP

RTP

RTP is the Real-time Transport Protocol. Here is an excerpt from the RTP specification: "RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulated data, over multicast or unicast network services. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services. The data transport is augmented by a control protocol (RTCP) to allow monitoring of the data delivery in a manner scalable to large multicast networks, and to provide minimal control and identification functionality. RTP and RTCP are designed to be independent of the underlying transport and network layers". For further information please refer to RFC 1889: RTP: A Transport Protocol for Real-Time Applications.

RTSP

RTSP is the Real Time Streaming Protocol. It is an application-level protocol for control over the delivery of real-time data, such as audio and video. Sources of data can include both live feed and stored clips. RTSP is the equivalent of the TV remote control for media streams. You can start and stop streams on the server side (video on demand type of applications). For furthur information please refer to RFC 2326: Real Time Streaming Protocol (RTSP).

Features of RTP supported by JMF

JMF supports both transmission and reception of media over RTP for a variety of media formats. It also fully supports RTCP. Besides, JMF implements the "RTP Profile for Audio and Video Conferences with Minimal Control", RTP/AVP. The front end GUI application, JMStudio can be used as a standalone application to transmit and receive RTP streams.

Features of RTSP supported by JMF

JMF supports RTSP on the client side for reception and playback. The RTSP protocol stack is built into the player which can be accessed through the Manager.createPlayer() interface with an RTSP media locator. JMF interoperates with other standard-based RTSP servers.

Swing (Java)

Swing is a GUI toolkit for Java. Swing is one part of the Java Foundation Classes (JFC). Swing includes graphical user interface (GUI) widgets such as text boxes, buttons, split-panes, and tables.

Swing widgets provide more sophisticated GUI components than the earlier Abstract Windowing Toolkit. Since they are written in pure Java, they run the same on all platforms, unlike the AWT which is tied to the underlying platform's windowing system. Swing supports pluggable look and feel – not by using the native platform's facilities, but by roughly emulating them. This means you can get any supported look and feel on any platform. The disadvantage of lightweight components is possibly slower execution. The advantage is uniform behavior on all platforms.

History

The Internet Foundation Classes (IFC) were a graphics library for Java originally developed by Netscape Communications Corporation and first released on December 16, 1996.

On April 2, 1996, Sun Microsystems and Netscape Communications Corporation announced their intention to combine IFC with other technologies to form the Java Foundation Classes. In addition to the components originally provided by IFC, Swing introduced a mechanism that allowed the look and feel of every component in an application to be altered without making substantial changes to the application code. The introduction of support for a pluggable look and feel allowed Swing components to emulate the appearance of native components while still retaining the benefits of platform independence.

Originally distributed as a separately downloadable library, Swing has been included as part of the Java Standard Edition since release 1.2. The Swing classes are contained in the javax.swing package hierarchy.

Relationship to AWT

Since early versions of Java, a portion of the Abstract Windowing Toolkit (AWT) has provided platform independent APIs for user interface components. In AWT, each component is rendered and controlled by a native peer component specific to the underlying windowing system.

By contrast, Swing components are often described as *lightweight* because they do not require allocation of native resources in the operating system's windowing toolkit. The AWT components are referred to as *heavyweight components*.

Much of the Swing API is generally a complementary extension of the AWT rather than a direct replacement. In fact, every Swing lightweight interface ultimately exists within an AWT heavyweight component because all of the top-level components in Swing (JApplet, JDialog, JFrame, and JWrindow) extend an AWT top-level container. The core rendering functionality used by Swing to draw its lightweight components is provided by Java2D, a part of AWT. However, the use of lightweight and heavyweight components within the same window is generally discouraged due to Z-order incompatibilities.

Relationship to SWT

The Standard Widget Toolkit (SWT) is a competing toolkit originally developed by IBM and now maintained by the Eclipse Foundation. SWT's implementation has more in common with the heavyweight components of AWT. This confers benefits such as more accurate fidelity with the underlying native windowing toolkit, at the cost of an increased exposure to the native resources in the programming model.

The advent of SWT has given rise to a great deal of division among Java desktop developers with many strongly favouring either SWT or Swing. A renewed focus on Swing look and feel fidelity with the native windowing toolkit in the approaching Java SE 6 release (as of 2006) is probably a direct result of this.

Example

The following is a Hello World program using Swing.

```
package helloworld;
import javax.swing.JFrame;
import javax.swing.JLabel;

public final class HelloWorld extends JFrame {
   private HelloWorld() {
      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
      add(new JLabel("Hello, World!"));
      pack();
      setLocationRelativeTo(null);
   }

   public static void main(String[] args) {
      new HelloWorld().setVisible(true);
   }
}
```

5. SYSTEM DESIGN

The most creative and challenging phase of the system life cycle is system design. The term design describes a final system and the process by which it is developed. It refers to the technical specifications that will be applied in implementing the candidate system. It also includes the construction of programs and program testing.

The first step in system designing is to determine how the output is to be produced and in what format. Samples of the output and input are also presented. In the second step, input data and master files are to be designed to meet requirement of the proposed output. The processing phases are handled through program construction and testing, including a list of the programs needed to meet the system's objectives and complete documentation.

Finally details related to justification of the system and an estimate of the impact of the candidate system on the user and the organization are documented and evaluated by management as a step towards implementation. The final report prior to the implementation phase includes procedure flow chart, record layouts, and a workable plan for implementing the candidate system.

System design has two phases:

1. Logical and
2. Physical.

The logical design reviews the present physical system, prepares the input and output and also prepares a logical design walk-through. We have to deal with how to take entries required and whether and how to process the user data. Also we have to deal with how to present the data in an informative and appealing format. This design also involves the methodology to store, modify and retrieve data from the database as per the requirement.

Physical design maps out the details of the physical system, plans the system implementations, devices a test and implementation plan and new hardware and software. We have to decide how and where to store the input data and how to process it so as to present it to the user in an easy, informative and attractive manner. A major step in the

design is the preparation of input and output reports in a form acceptable to the user. In this a data entry operator can feed the relevant details asked by the system for a particular task as input. The system will store the data in computer files and can be processed when needed.

5.1 Input Design

The input design is one of the important tasks in the software development, since it helps to reduce the user's work and select the correct data entry. The software includes various screens, which helps to accept the correct entry The administrator-gathered and user-entered inputs are converted into a computer-based format. It also includes determining the record media, method of input, speed of capture, and entry into the screen. Inputs to the system has to be both through the movement of the mouse and also via the keyboard. The frame/pages requested can be viewed on a Cathode Ray Tube (CRT) screen since the system under development is a general Web Portal the data input is to be done from the Administrative account.

Login Form:

It provides security to the system. This allows users to use the system, by entering username and IP address.

5.2 Output Design

Output is the most important and direct source in information to the consumer and Administrator. Intelligent output design will improve the systems relationship with user and help in decision making.The output is designed in such a way that it is attractive, convenient and informative. As the outputs are the most important sources of information to the user, better design should improve the user's relationship with us.

Capturing still images, videos and sound enhances the appearance and attractiveness of the output. The output is Pleasing to the user and conveys information in an appealing manner. The output has to be such that the user shall find the required information easily. Unnecessary clutter is avoided except for a few features which are required to improve appearance.

The output is provided in a categorized manner. Items of similar nature are grouped together. For instance in the News field, there can be several subcategories like Politics, Sports, Business and so on. The features which users might be more interested in are highlighted in separate columns along the sides of the page. We have provided following interfaces for output purposes:

- Interface for the Group Conversation Window
- Interface for the Private Messaging.
- Interface for recording and playing audio file.

5.3 Module wise Description

This project has the following modules.
They are:

- Text Module
- Audio Module
- Video Module
- Recorder Player Module

The functions of each module are as follows.

5.3.1 Text Module

In this module we have provided user with the facility to send text messages to any other online user. We have provided the facility of private messaging between two users as well as the user can have conference with more than one users.

5.3.2 Audio Module

In this module we have provided user with the facility to have audio conversation with any other online user. The user can have audio chat using their microphones.

5.3.3 Video Module

In this module we have provided user with the facility to have video conversation with any other online user. The users can capture live video pictures using web cams and send it in the form of streams to the other user.

5.3.4 Recorder-Player Module

This module provides the facility to record voice from microphone and store it as an audio file. The user can also specify the different formats into which to record the audio file like wave, aifc, au etc. The user has been given options to start and end the recording.

The player module will play the recorded audio file from the path specified by the user.

5.4 Data Flow Diagrams

As information is moved through the software, it is modified by a series of transformations. A data flow diagram is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output**.** The data flow diagram may be used to represent a system or software at any level of abstraction. DFD's may be portioned into levels that represent increasing information flow and functional detail.

A level 0 DFD, also called a fundamental system model or a context model, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows respectively. Each of the processes represented at the level 1 is a sub function of the overall system depicted in the context model.

Data Flow Diagram is a way of expressing system requirements in a graphical form. It has the purpose of identifying major transformation that will become programs in system design. DFD does not supply detailed description of the modules but graphically describes a system's data and how the data interact with the system.

A Data Flow Diagram (DFD) or a Bubble chart is a graphical tool for structured analysis, it was De Marco (1978) and Gane and Sarson (1979) who introduced DFD. DFD models

a system by using external entities from which data flows to a process, which transforms the data and creates, output-data-flows which go to other processes or external entities or files. Data in files may also flow to processes as inputs.

There are various symbols used in a DFD. Bubbles represent the processes. Named arrows indicate the data flow. External entities are represent by rectangles and are outside the system such as vendors or customers with whom the system interacts. They either supply or consume data. Entities supplying data are known as sources and those that consume data are called sinks. Data are stored in a data store by a process in the system. Each component in a DFD is labeled with a descriptive name. Process names are further identified with a number. DFD's can be hierarchically organized, which help

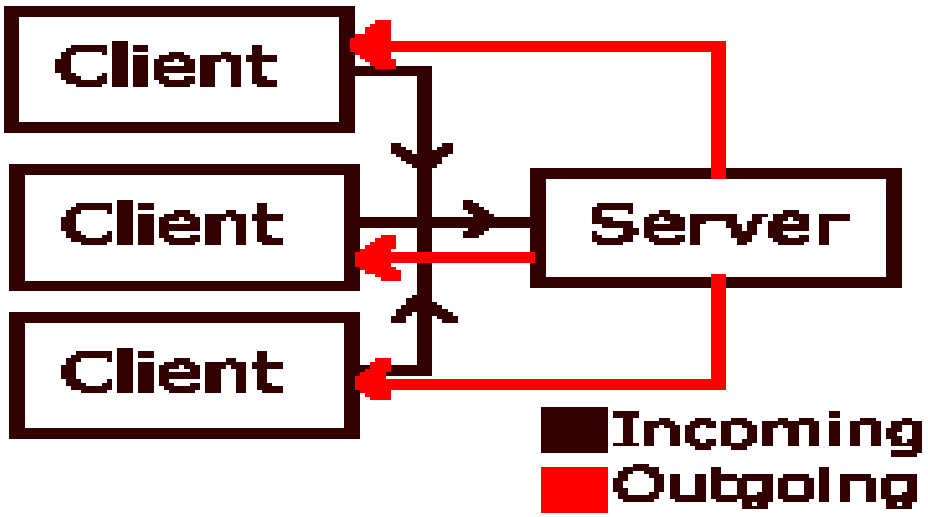
in partitioning and analyzing large systems. As a first step, one Data Flow Diagram can depict an entire system, which gives the system overview. It is called Context Diagram of level 0 DFD. The Context Diagram can be further expanded. The successive expansion of a DFD from the context diagram to those giving more details is known as leveling of DFD. Thus a top down approach is used, starting with an overview and then working out the details

The main merit of DFD is that it can provide an overview of what data a system would process, what transformation of data are done, what files are used, and where the results flow.

The server application will listen for any new connections from client connections on port 1000. Any new client connections will be added to an internal stack of client connections. All client applications will only communicate to the server and not to other client computers. This ensures that a client system cannot show the remote address of a particular remote user and results in improved security against other users of the application.

Figure shows the data flow of the application between the clients and server. Each client will have a two-way communication link (send and receive) with the server. The server is responsible for relaying any messages between clients. The server may also log any events and data to a local file on the server computer.

Figure– Data Flow of Application



5.5 Data Dictionary

A Data Dictionary is a repository of various data flows defined in a DFD. The associated data dictionary states precisely the structure of each data flow in the DFD. Components in the structure of a data flow may also be specified in the data dictionary. To define the data structure various notations of regular expressions are used.

Sl. No	Code	Description
1.	Login ID	User Name
2.	Server Address	Server's IP Address
3.	Server Port	Server's Port Number

6. SYSTEM IMPLEMENTATION

An important aspect of the system analyst job is to make sure that the design is implemented to establish standards. Implementation invoices the conversion of basic application to complete replacement of computer system. It is a process of converting a new revised system design into an operational one. It is simply a translation of the largest abstraction into physical realization, using language architecture.

Implementation includes all the activities that take place to convert the old system to new System may be totally new, replacing an existing manual or automated system or it may be a proper implementation essential to provide a reliable system to meet organizations equipment**.**

6.1 Implementation Aspect

In the implementation of a computer system to replace a manual system, the problems encountered are converting files, training users, creating accurate files and verifying print outs for integrity. Implementation of a new computer system to replace an existing one is more difficult conversion. If not properly planned, there can be many problems. Some larger computer systems have taken long years to convert.

Implementation of modified application to replace an existing system using the same computer is relatively easy to handle provided that there are no major changes in the file. Implementation is a key stage in achieving a successful new system, because it usually involves a lot of upheaval in the user department. During the design phase the product structure, its undergoing data structures, the general algorithms and interfaces and linkage among the various substructures are established. The algorithms and data structures developed during design based on requirement specifications were converted to running programs.

6.2 Post implementation review

After system implementation and user training is complete a review of the system is usually conducted by the users and the analyst. This is a formal process to determine how well the system is working. How it has been expected and whether adjustments are made. The review is also important to gather information for maintenance of the system. The most fundamental concern during post implementation review is determined whether the system has met its objectives. The performance level of users is reviewed and checked whether the system is producing the indented results. The systems output quality merits special attention. The accuracy of information, the timelessness of 0presentation, completeness and appropriateness of formats etc. Continue to indicate system quality. In some cases unsuitable system component may found during post implementation review. The post implementation review not only assesses how well the current system of design is implemented but also a valuable thought information review.

6.3 Sample Codes

6.3.1 To send voice over LAN(Server Program)

```
import java.io.File;
import java.io.IOException;
import java.io.*;
import java.net.*;
import javax.sound.sampled.*;

public class Server {

public static void main(String[] args) {

   float sampleRate = 8000.0F; //possible 8000 11025 16000 22050 44100
   int sampleSizeInBits = 16; //possible 8 or 16
   int channels = 1; //1 or 2
```

```
   boolean signed = true;
   boolean BigEndian = false;
     AudioFormat audio = new AudioFormat(sampleRate, sampleSizeInBits, channels, signed, BigEndian);
   try {
    TargetDataLine targetDataLine;
    DataLine.Info dataLineInfo = new DataLine.Info(TargetDataLine.class,audio);
    //targetDataLine = (TargetDataLine) AudioSystem.getLine(dataLineInfo);
    //Wait for a Socket Connection
    ServerSocket server = new ServerSocket(5656);
    while (true) {
     Socket client = server.accept();
     Thread c = new CaptureThread(client, audio, dataLineInfo);
     c.start();
    }
   }
   catch (Exception e) {System.out.println(e);}
  }

}
class CaptureThread extends Thread {
  Socket client;
  TargetDataLine targetDataLine;
  AudioFormat audio;
  DataLine.Info info;
  public CaptureThread(Socket sock, AudioFormat aud, DataLine.Info dataLineInfo) {
   client = sock;
   audio = aud;
   info = dataLineInfo;
  }
```

```
  public void run() {
    AudioFileFormat.Type fileType = null;
    fileType = AudioFileFormat.Type.WAVE;
    int numBytesRead = 0;
    byte[] data = new byte[1024];
BufferedOutputStream out = null;
    try {
      out = new BufferedOutputStream(client.getOutputStream());
    } catch (Exception ear) {System.out.println("Bufferedout " + ear);}
    try {
      System.out.println("Got to setting targetDataline");
      targetDataLine = (TargetDataLine) AudioSystem.getLine(info);
      System.out.println(targetDataLine.getBufferSize());
      targetDataLine.open(audio, targetDataLine.getBufferSize());
      System.out.println("Opened The dataLine");
      targetDataLine.start();
      System.out.println("Started TargetData");
    } catch (Exception ex) {System.out.println(ex);}

      while (true) {
        try {
          numBytesRead = targetDataLine.read(data, 0, 1024);
          out.write(data, 0, numBytesRead);
        }
        catch (Exception eat) {System.out.println("Error Stream " + eat);}
      }
    }
  }
```

6.3.2 To Send Voice Over LAN(Client Program)

```
import java.io.*;
import javax.sound.sampled.*;
import java.net.*;

public class client {
 //public client() throws Exception{
   public static void main(String[] args) throws Exception {
   boolean thread = false;
   Socket client = new Socket("localhost", 5656);
                    //ObjectInputStream in = new ObjectInputStream(new
BufferedInputStream(client.getInputStream()));
   Thread c = new downthread(client);
   c.start();
   }
}

class downthread extends Thread {
Socket client;
boolean thread = true;

public downthread(Socket clt) {
 client = clt;


   }
 public void run() {
  try {
    final int bufSize =16384;
    SourceDataLine line;
    BufferedInputStream playbackInputStream;
    float sampleRate = 8000.0F; //possible 8000 11025 16000 22050 44100
```

```
            int sampleSizeInBits = 16; //possible 8 or 16
            int channels = 1; //1 or 2
            boolean signed = true;
            boolean BigEndian = false;
               AudioFormat format = new AudioFormat(sampleRate, sampleSizeInBits,
channels, signed, BigEndian);

                              playbackInputStream = new BufferedInputStream(new
AudioInputStream(new                    BufferedInputStream(client.getInputStream()),
format,564564564));
            DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
            line = (SourceDataLine) AudioSystem.getLine(info);
            line.open(format, bufSize);
            byte[] data = new byte[1024];
            int numBytesRead = 0;
            line.start();

            while (thread != false) {
              numBytesRead = playbackInputStream.read(data);
              line.write(data, 0, numBytesRead);

             }
           }
           catch (Exception e) {System.out.println(e);}
         }

       }
```

6.3.3 To Capture Video Streams Using Web Cam

```
import java.io.*;
import javax.media.*;
import javax.media.control.*;
import javax.media.datasink.*;
import javax.media.format.*;
import javax.media.protocol.*;
public class TestQuickCamPro
{
        private static boolean  debugDeviceList = false;

        private static String defaultVideoDeviceName = "Microsoft WDM Image Capture";
        private static String defaultAudioDeviceName = "DirectSoundCapture";
        private static String    defaultVideoFormatString= "size=176x144, encoding=yuv, maxdatalength=38016";
        private static String defaultAudioFormatString = "linear, 16000.0 hz, 8-bit, mono, unsigned";
        private static CaptureDeviceInfo      captureVideoDevice = null;
        private static CaptureDeviceInfo      captureAudioDevice = null;
        private static VideoFormat            captureVideoFormat = null;
        private static AudioFormat            captureAudioFormat = null;

        public static void main(String args[])
        {
                for (int x = 0; x < args.length; x++)
                {
                                if (args[x].toLowerCase().compareTo("-dd") == 0)
                                debugDeviceList = true;
                }
```

```
            Stdout.log("get list of all media devices ...");
        java.util.VectordeviceListVector= CaptureDeviceManager.getDeviceList(null);
            if (deviceListVector == null)
            {
                Stdout.log("... error: media device list vector is null, program
aborted");
                System.exit(0);
            }
            if (deviceListVector.size() == 0)
            {
                Stdout.log("... error: media device list vector size is 0, program
aborted");
                System.exit(0);
            }

            for (int x = 0; x < deviceListVector.size(); x++)
            {
                // display device name
                CaptureDeviceInfo deviceInfo = (CaptureDeviceInfo)
deviceListVector.elementAt(x);
                String deviceInfoText = deviceInfo.getName();
                if (debugDeviceList)
                    Stdout.log("device " + x + ": " + deviceInfoText);

                // display device formats
                Format deviceFormat[] = deviceInfo.getFormats();
                for (int y = 0; y < deviceFormat.length; y++)
                {
                    // serach for default video device
                    if (captureVideoDevice == null)
                        if (deviceFormat[y] instanceof VideoFormat)
```

```
                    if
(deviceInfo.getName().indexOf(defaultVideoDeviceName) >= 0)
                {
                    captureVideoDevice = deviceInfo;
                    Stdout.log(">>> capture video device = " +
deviceInfo.getName());
                }

                // search for default video format
                if (captureVideoDevice == deviceInfo)
                    if (captureVideoFormat == null)
                    if
(DeviceInfo.formatToString(deviceFormat[y]).indexOf(defaultVideoFormatString) >= 0)
                {
                    captureVideoFormat = (VideoFormat)
deviceFormat[y];
                    Stdout.log(">>> capture video format = " +
DeviceInfo.formatToString(deviceFormat[y]));
                }

                // serach for default audio device
                if (captureAudioDevice == null)
                    if (deviceFormat[y] instanceof AudioFormat)
                    if
(deviceInfo.getName().indexOf(defaultAudioDeviceName) >= 0)
                {
                    captureAudioDevice = deviceInfo;
                    Stdout.log(">>> capture audio device = " +
deviceInfo.getName());
                }
```

```
                        // search for default audio format
                        if (captureAudioDevice == deviceInfo)
                                if (captureAudioFormat == null)
                                if
(DeviceInfo.formatToString(deviceFormat[y]).indexOf(defaultAudioFormatString) >= 0)
                        {
                                captureAudioFormat = (AudioFormat)
deviceFormat[y];
                                Stdout.log(">>> capture audio format = " +
DeviceInfo.formatToString(deviceFormat[y]));
                        }

                        if (debugDeviceList)
                                Stdout.log(" - format: " +
DeviceInfo.formatToString(deviceFormat[y]));
                }
        }
        Stdout.log("... list completed.");
        if (debugDeviceList)
                System.exit(0);
        MediaLocator videoMediaLocator = captureVideoDevice.getLocator();
        DataSource videoDataSource = null;
        try
        {
                videoDataSource =
javax.media.Manager.createDataSource(videoMediaLocator);
        }
        catch (IOException ie) { Stdout.logAndAbortException(ie); }
        catch (NoDataSourceException nse)
{ Stdout.logAndAbortException(nse); }
```

```
            if (! DeviceInfo.setFormat(videoDataSource, captureVideoFormat))
            {
                  Stdout.log("Error: unable to set video format - program aborted");
                  System.exit(0);
            }
            MediaLocator audioMediaLocator = captureAudioDevice.getLocator();
            DataSource audioDataSource = null;
            try
            {
                  audioDataSource =
javax.media.Manager.createDataSource(audioMediaLocator);
            }
            catch (IOException ie) { Stdout.logAndAbortException(ie); }
            catch (NoDataSourceException nse)
{ Stdout.logAndAbortException(nse); }

            if (! DeviceInfo.setFormat(audioDataSource, captureAudioFormat))
            {
                  Stdout.log("Error: unable to set audio format - program aborted");
                  System.exit(0);
      }
            DataSource mixedDataSource = null;
            try
            {
                  DataSource dArray[] = new DataSource[2];
                  dArray[0] = videoDataSource;
                  dArray[1] = audioDataSource;
                  mixedDataSource=
javax.media.Manager.createMergingDataSource(dArray);
            }
```

```
            catch(IncompatibleSourceExceptionise){
Stdout.logAndAbortException(ise); }
FileTypeDescriptoroutputType=new FileTypeDescriptor(FileTypeDescriptor.MSVIDEO);
            Format outputFormat[] = new Format[2];
            outputFormat[0] = new VideoFormat(VideoFormat.INDEO50);
            outputFormat[1] = new AudioFormat(AudioFormat.GSM_MS /* LINEAR
*/);
                        ProcessorModelprocessorModel=new
ProcessorModel(mixedDataSource, outputFormat, outputType);
            Processor processor = null;
            try
            {
                  processor = Manager.createRealizedProcessor(processorModel);
            }
            catch (IOException e) { Stdout.logAndAbortException(e); }
            catch (NoProcessorException e) { Stdout.logAndAbortException(e); }
            catch (CannotRealizeException e) { Stdout.logAndAbortException(e); }

            DataSource source = processor.getDataOutput();
            MediaLocator dest = new MediaLocator("file:testcam.avi");
            DataSink dataSink = null;
            MyDataSinkListener dataSinkListener = null;
            try
            {
                  dataSink = Manager.createDataSink(source, dest);
                  dataSinkListener = new MyDataSinkListener();
                  dataSink.addDataSinkListener(dataSinkListener);
                  dataSink.open();
            }
            catch (IOException e) { Stdout.logAndAbortException(e); }
            catch (NoDataSinkException e) { Stdout.logAndAbortException(e); }
```

```
                catch (SecurityException e) { Stdout.logAndAbortException(e); }
                try
                {
                        dataSink.start();
                }
                catch (IOException e) { Stdout.logAndAbortException(e); }
                processor.start();
                Stdout.log("starting capturing ...");
                try { Thread.currentThread().sleep(10000); } catch (InterruptedException
ie) {}
                Stdout.log("... capturing done");
                processor.stop();
                processor.close();
                dataSinkListener.waitEndOfStream(10);
                dataSink.close();
                Stdout.log("[all done]");
        }
}
```

7. SYSTEM TESTING

Testing and Implementation are the final and important phase. It involves user training, system testing and successful running of the developed proposed system. The user tests the developed system and changes are made according to their needs. The testing phase involves the testing of developed system using various kinds of data.
An elaborate testing of data is prepared and the system is tested using that test data. While testing, errors are noted and corrections are being made. The users are trained to operate the developed system successfully in future.

7.1 Testing

System testing is the stage of implementation, which is aimed at ensuring that the system works accurately and efficiently before live operation commences. Testing is vital to the success of the system. System testing makes a logical assumption that if all the parts of the system are correct, the goal will be successfully achieved. The candidate system is subject to a variety of tests: online response, volume, stress, recovery and security and usability tests. A series of testing are performed for the proposed system before the system is ready for user acceptance testing.

The testing steps are: -

- Unit Testing
- Integration Testing
- Validation
- Output Testing
- User Acceptance Testing

7.1.1 Unit Testing

Unit testing involves verification efforts on the smallest unit of software design the module. This is also known as “Module Testing”. The modules are tested separately. This testing is carried out during programming stage itself. In this testing step, each

module is found to be working satisfactorily as regard to the expected output from the module. Unit testing is normally considered and adjunct to the coding step. After source level code has been developed, reviewed and verified for correct syntax, unit test case design case design begins. Each test case should be coupled with a set of expected results. Unit testing is simplified when modules with high cohesion is designed. When a module a number of test cases only addresses only one function is reduced and error can be more easily uncovered.

The test that occurs as a part of unit testing is the model interface is tested to ensure that information correctly flows into and out of the program unit under test. The local data structure is examined to ensure that data store temporarily maintains its integrity during all steps of algorithm execution.

Boundary conditions are tested to ensure that modules operate properly. All independent paths through the control structure exercised to ensure that all statements in a module have been executed at least once. And finally, all error-handling paths are tested.

7.1.2 Integration Testing

Data can be lost across an interface; on module can have an adverse effort on other sub functions, when combined, may not produce the desired major functions. Integration testing is a systematic testing for constructing the program structure, while at the same time conducting tests to uncover errors associated with the interface. The objective is to take unit tested modules and build a program structure. All the modules are combined and tested as a whole. Here correction is difficult because the vast expenses of the entire program complicate the isolation of causes. Thus, in the integration-testing step, all the errors uncovered are corrected for the next testing steps.

7.1.3 Output Testing

After performing the validation testing, the next step is output testing of the proposed system since no system could be useful if it does not produce the required output in the specific format. Asking the users about the format required by them tests the

outputs generated or displayed by the system under consideration. Here, the output format is considered in two ways: - one is on screen and another is printed format.
The output format on the screen is found to be correct as the format was designed in the system design phase according to the user needs. For the hardcopy also, the output comes out as the specified requirements by the user. Hence, output testing does not result in any correction in the system.

7.1.4 User Acceptance Testing

User acceptance of a system is the key factor for the success of any system. The system under consideration is tested for user acceptance by constantly keeping in touch with the prospective system users at time of developing and making changes wherever required. This is done with regard to the following points: -

- Input screen design
- Output screen design
- On-line message to guide the user

The above testing, are done by taking various kinds of test data. Preparation of test data plays a vital role in the system testing. After preparing the test data, the system understudy is tested using the test data. While testing the system by using test s\data errors are again uncovered and corrected by using above testing steps and corrections are also noted for future use.

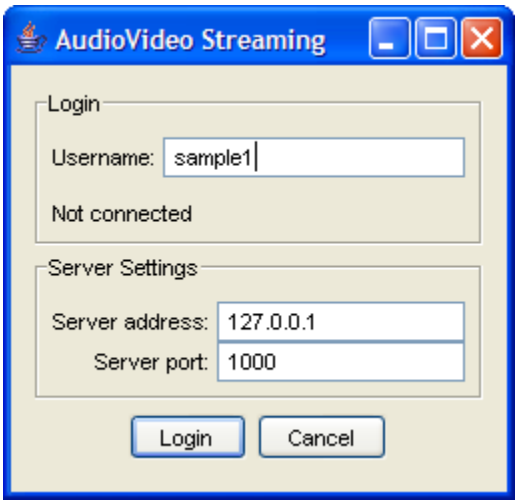
7.2 User training

After the system is implemented successfully, training of the user is one of the most important subtasks of the developer. For the purpose user system manuals are prepared and handed over to the user to operate the developed system. Both the hardware and software securities are made to run the developed systems successfully in future.

8. SNAPSHOTS

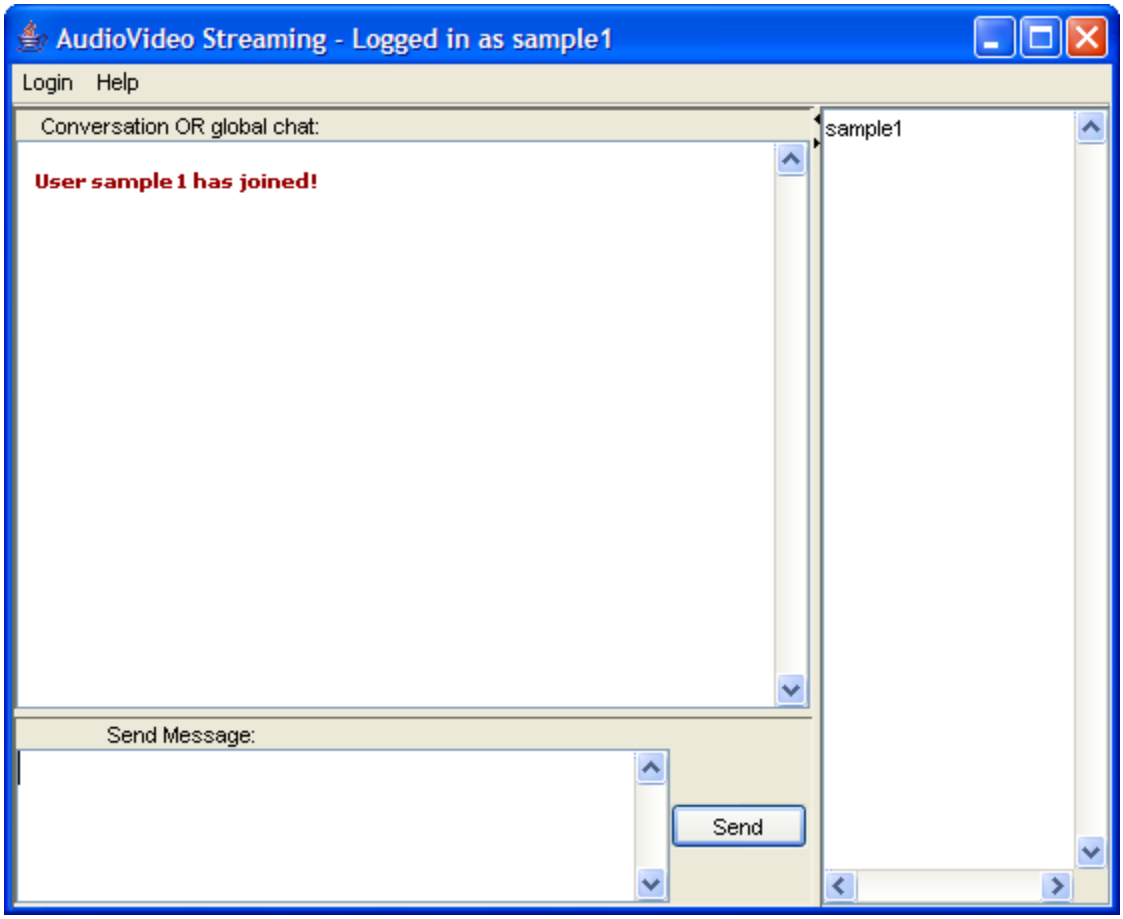
8.1 Interface for the Login Page

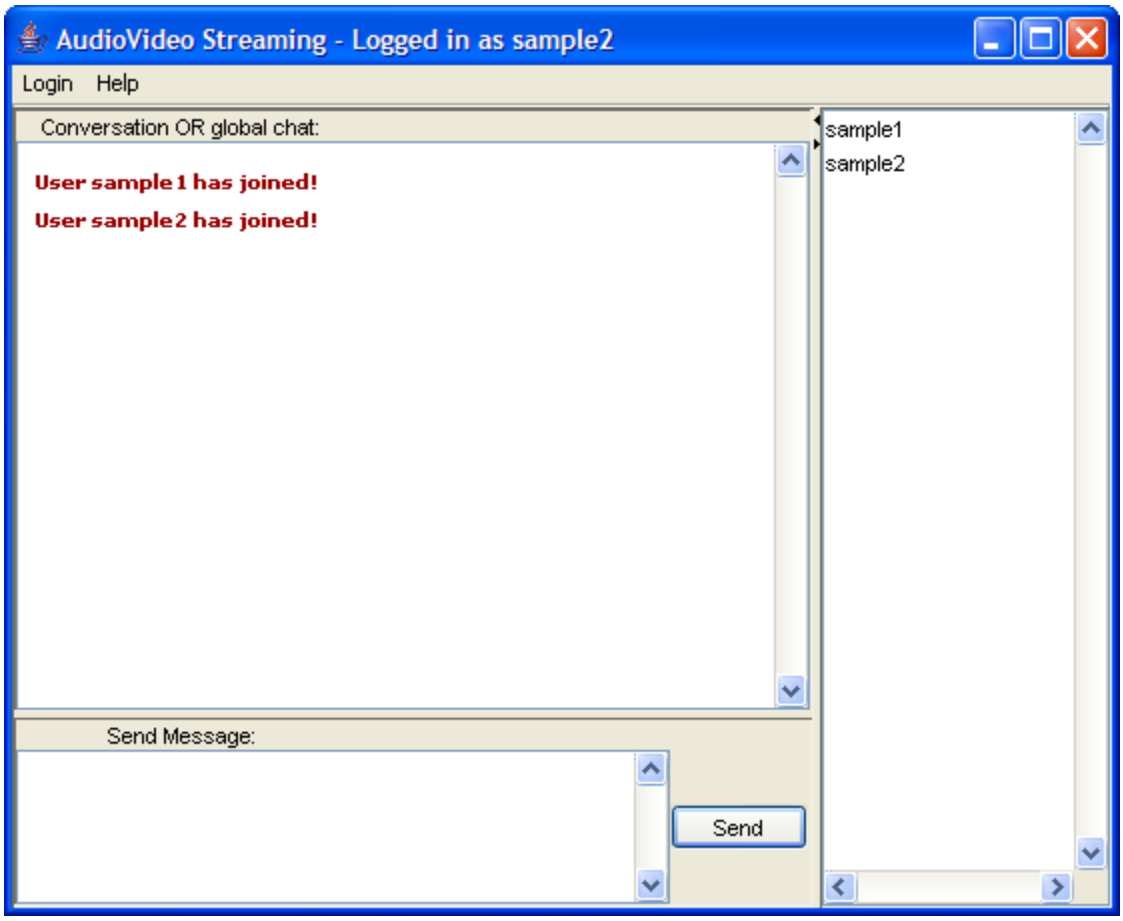
This interface allows the client to login by providing a user name ,server's IP addres and port number.

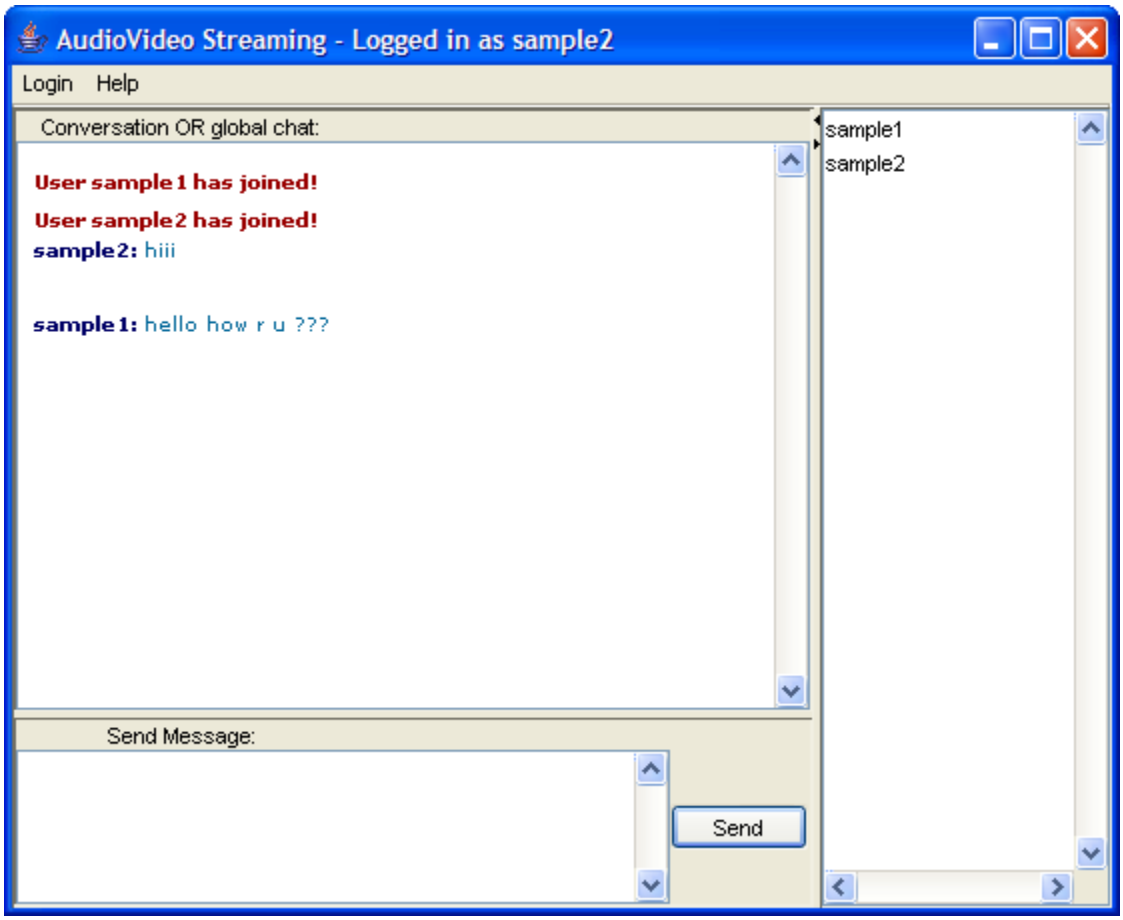


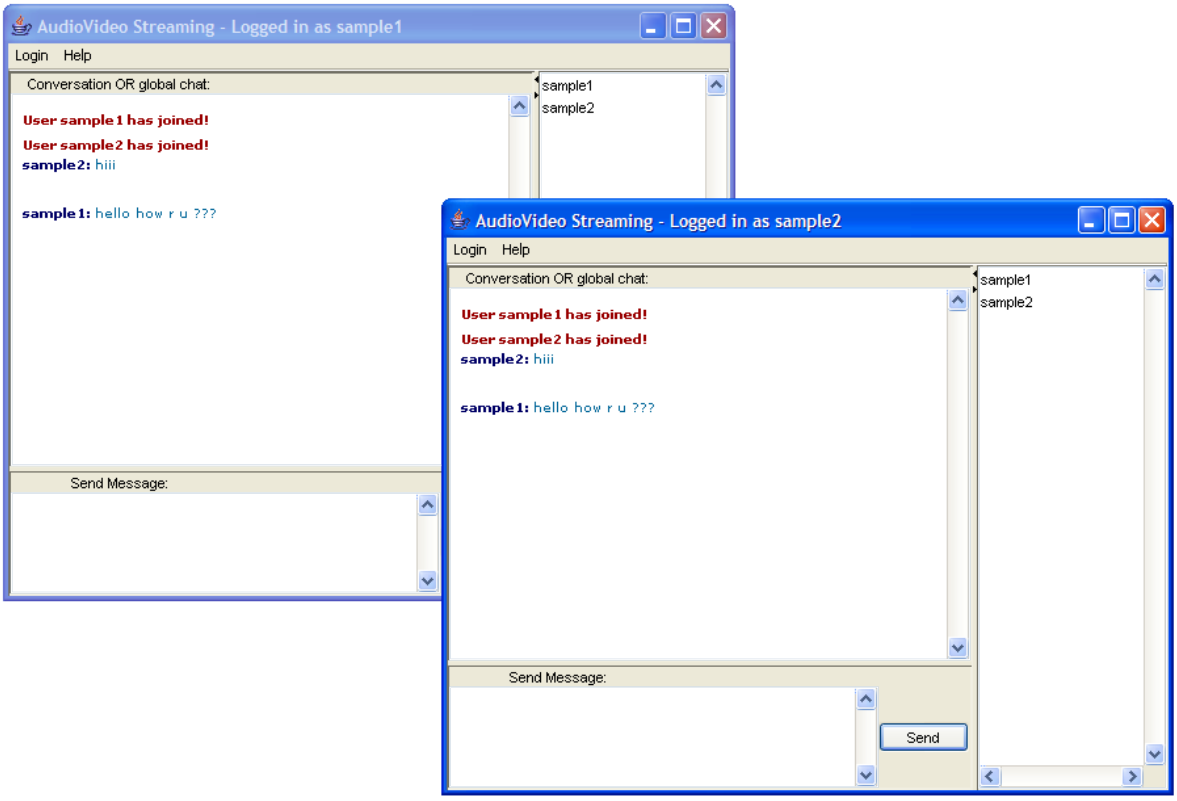
8.2 Interface for the Group Conversation Window

This is the group conversation window which allows multiple client to have a group conversation.



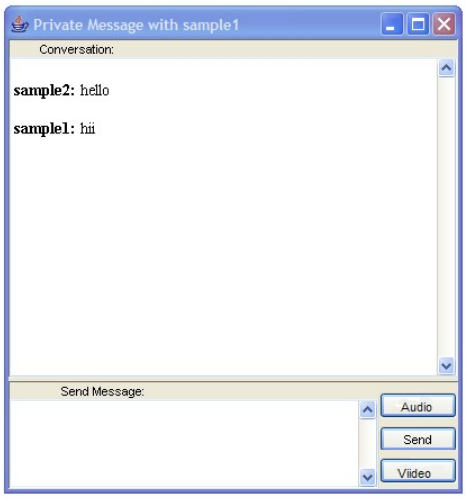

AudioVideo Streaming - Logged in as sample2
Login
Help
Conversation OR global chat:
User sample1 has joined!
User sample2 has joined!
sample1
sample2
Send Message:
Send

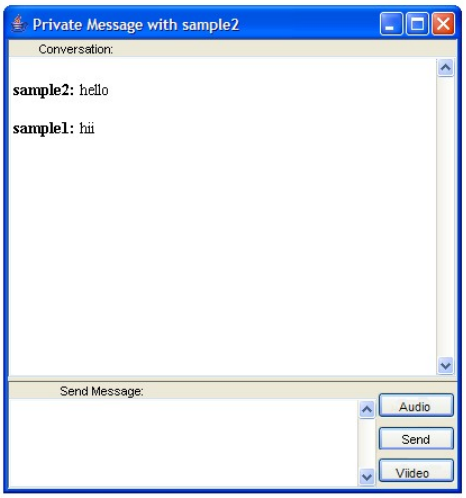

AudioVideo Streaming - Logged in as sample2
Login
Help
Conversation OR global chat:
User sample1 has joined!
User sample2 has joined!
sample2: hiii
sample1: hello how r u ???
sample1
sample2
Send Message:
Send


AudioVideo Streaming - Logged in as sample1
Login Help
Conversation OR global chat:
User sample1 has joined!
User sample2 has joined!
sample2: hiii
sample1: hello how r u ???
sample1
sample2
Send Message:
AudioVideo Streaming - Logged in as sample2
Login Help
Conversation OR global chat:
User sample1 has joined!
User sample2 has joined!
sample2: hiii
sample1: hello how r u ???
sample1
sample2
Send Message:
Send

8.3 Interface for the Private Messaging

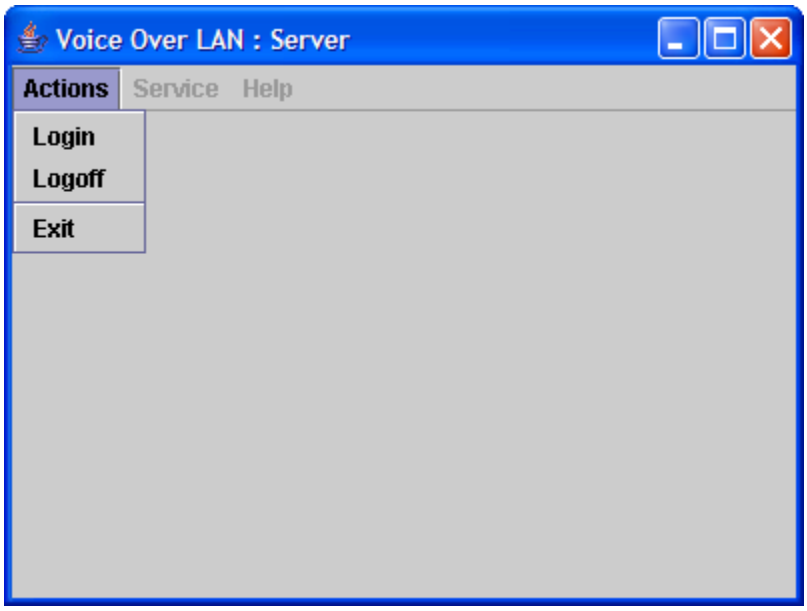
This is the private message window which allows two clients to communicate with each other in private.

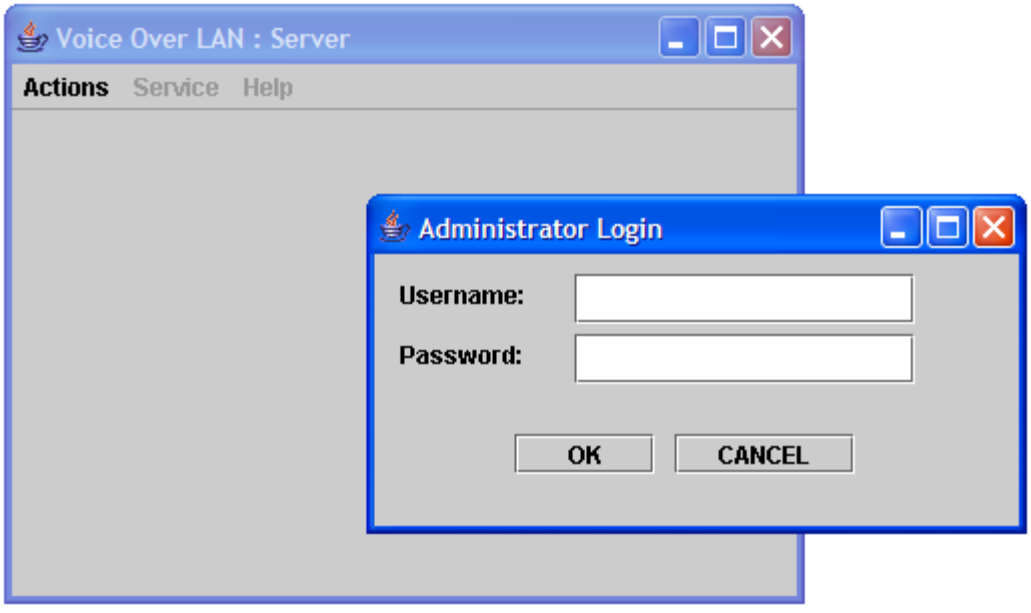


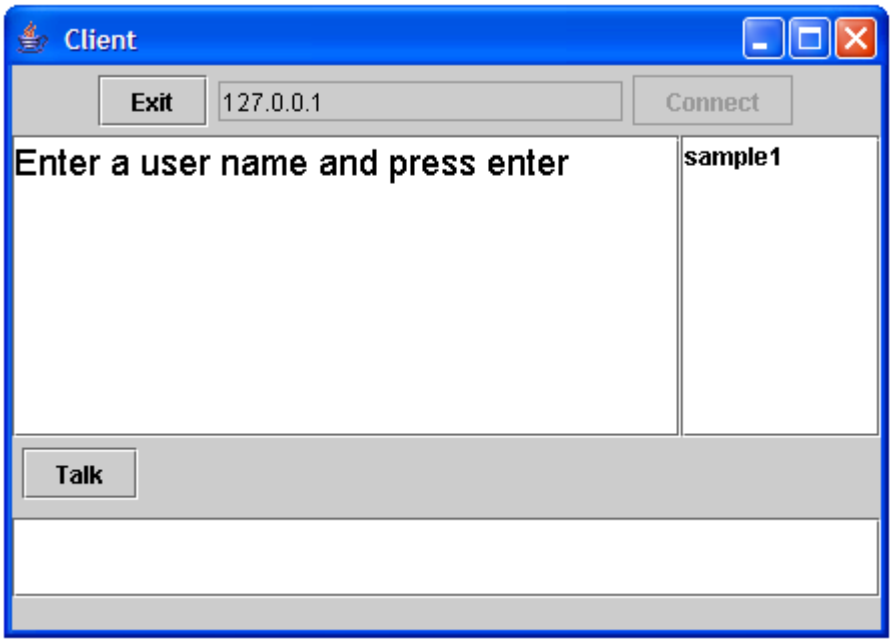


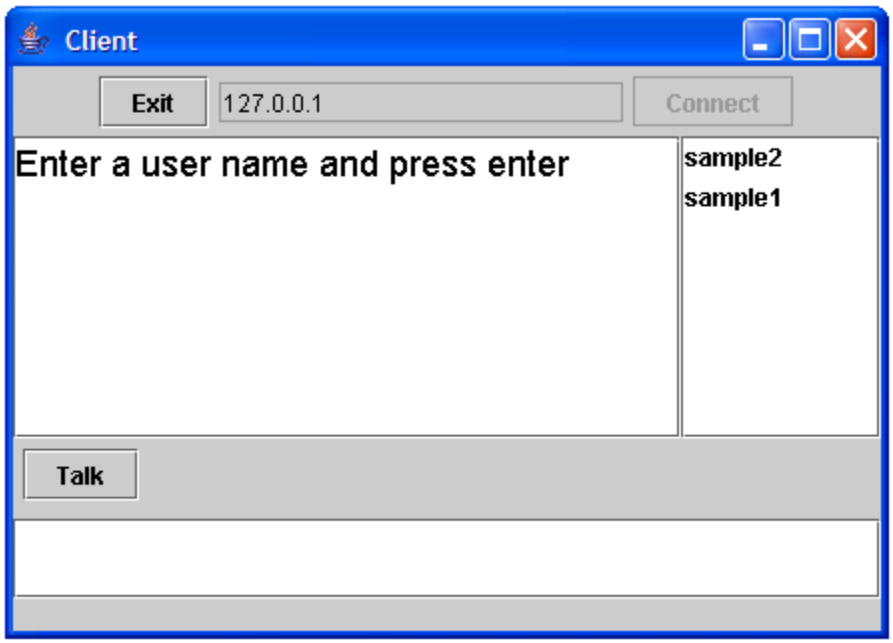
8.4 Interfaces for Audio Streaming

There is provision for administrator to start the audio server by providing username and password. Once the server is started the clients can have audio conversation by connecting to the server through its IP address.






Client
Exit
127.0.0.1
Connect
Enter a user name and press enter
sample1
Talk

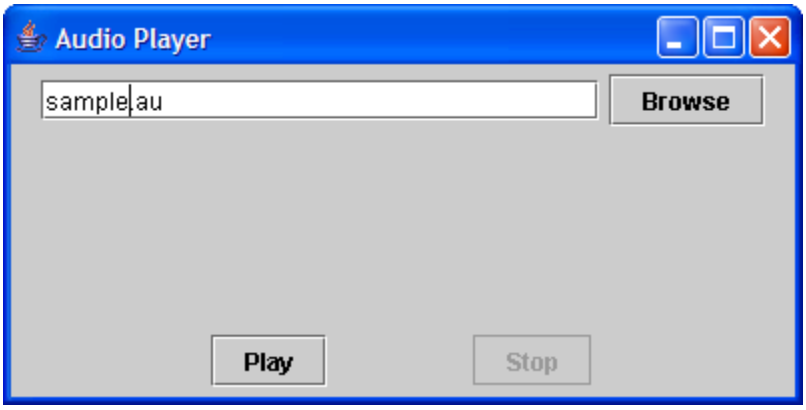

Client
Exit
127.0.0.1
Connect
Enter a user name and press enter
sample2
sample1
Talk

8.5 **Interface for Video Streaming**

This window displays the video streams captured by web cam connected to the clients system.

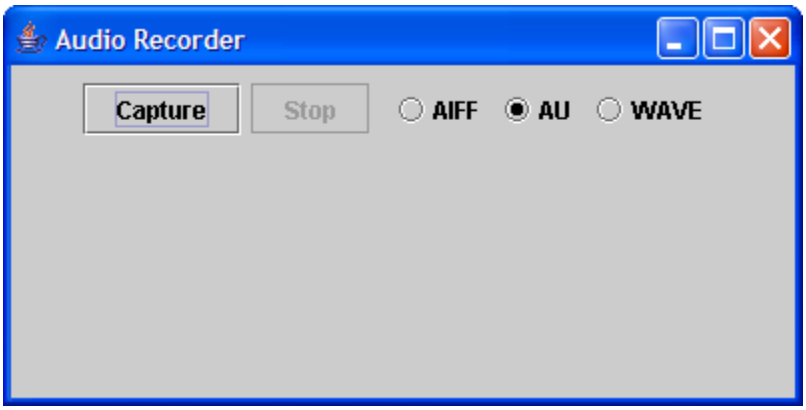
8.6 Audio Player

This interface provides the client with the option to play the recorded audio



8.7 Audio Recorder

This interface provides the client with the option to record audio.



9. FUTURE ENHANCEMENT

To survive from the competition each system has to produce some modifications to it in the future. New features will provide the system a new fresh look, by which it can attract a lot of users.

Following the popularity of the Internet and multimedia services over the Internet in recent years, soft videophones using Session Initiation Protocol (SIP) have become one of the major IP teleconference applications for desktops, or laptops, or handhold PCs. One of the soft videophone challenges is to be platform independent. This project can also be migrated to other OS like NOVELL NETWARE, OS/2 and other processors like ALPHA AXP, MIPS 4X00 series. This project can be enhanced to work on heterogeneous networks using RMI technology.

10. CONCLUSION

This project is intended to use the full features of the technological Revolution in the Current application development Scenario. Following the popularity of the Internet and multimedia services over the Internet in recent years, soft videophones using Session Initiation Protocol (SIP) have become one of the major IP teleconference applications for desktops, or laptops, or handhold PCs.

This is the best project in the world. This project uses the platform independent feature and JMF package of JAVA. The Java Media Framework API (JMF) enables audio, video and other time-based media to be added to applications and applets built on Java technology. This optional package, which can capture, playback, stream, and transcode multiple media formats, extends the Java 2 Platform, Standard Edition (J2SE) for multimedia developers by providing a powerful toolkit to develop scalable, cross-platform technology

11. BIBLIOGRAPHY

11.1 Textual References:

1. Java 2: The Complete Reference,Fifth Edition- Herbert Schildt
2. *"Teach Yourself JAVA in 21 Days" by Charles L. Perkins and Michael Morrison*
3. Orielly," Java Reference Library"
4. Java Cookbook by Ian Darwin Publisher: O'Reilly First Edition June 2001

11.2 Websites referred: -

1. http://www.google.com/
2. http://java.sun.com/products/java-media/jmf/
3. http://forum.java.sun.com/jmf/
4. http://en.wikipedia.org/wiki/Main_Page/jmf
5. http://www-128.ibm.com/developerworks/edu/j-dw-javajmf-i.html
6. http://java.sun.com/products/java-media/jmf/2.1.1/faq-jmf.html
7. http://java.sun.com/developer/technicalArticles/Media/JavaSoundAPI/