

Library copy  
Jan 11 1998

# DESIGNING OF APPLLET FOR CLIENT SERVER INFORMATION & CHAT SERVER PROTOCOL

*Dissertation submitted in partial fulfillment of the requirements for the  
award of the degree of*

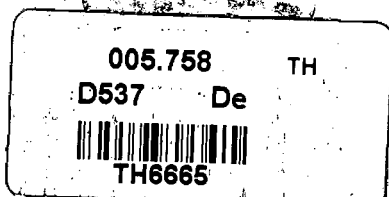
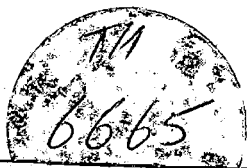
MASTER OF TECHNOLOGY  
*in*  
COMPUTER SCIENCE

*by*

**DEBENDRA KUMAR DHIR**



SCHOOL OF COMPUTER AND SYSTEM SCIENCES  
JAWAHARLAL NEHRU UNIVERSITY  
NEW DELHI - 110 067  
JANUARY, 1998



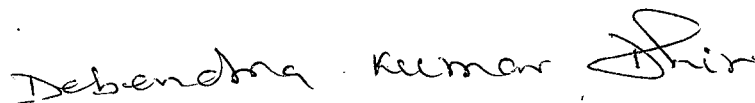
## CERTIFICATE

This is to certify that the dissertation entitled *Designing of APPLET for client server application and chat server protocol* being submitted by **Debendra Kumar Dhir** to the school of computer and system sciences, Jawaharlal Nehru University, New Delhi, in partial fulfilment of the requirements for the award of degree of *Master of technology* in Computer science is a bonafied work carried by him under the guidance and supervision of **Prof R.C. Phoha**.

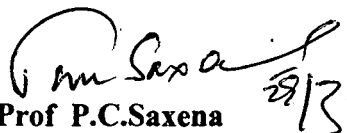
The matter embodied in the dissertation has not been submitted for the award of any other degree or diploma.



**Prof R.C. Phoha**  
SC&SS  
Jawaharlal Nehru University  
New Delhi 110067



**Debendra kumar Dhir**



**Prof P.C.Saxena**  
Dean SC&SS  
JAWAHARLAL NEHRU UNIVERSITY  
New Delhi 110067



## ACKNOWLEDGEMENT

The succes of this project has largely been due to the invaluable guidance of *Prof R.C Phoha* , my supervisor. My profound thanks to him for his helpful suggestions and motivation through out my work. I would like to thank *Prof P.C. Saxena Dean SC&SS*, for provinding the necessary facilities to complete this project.

I sincerely thank all my friends and faculty for thair insightful comments and help.



*Debendra Kumar Dhir*

**DEBENDRA KUMAR DHIR**

# CONTENTS

## CHAPTER 1.

### INTRODUCTION

- 1.1 Servers
- 1.2 Clients
  - 1.2.1 Operators
- 1.3 Channels
  - 1.3.1 Channel Operators
- 1.4 The IRC Specification.
  - 1.4.1 Overview
  - 1.4.2 Character codes
  - 1.4.3 Messages
    - 1.4.3.1 Messages format in pseudo BNF
  - 1.4.4 Numeric replies
- 1.5 IRC Concepts
  - 1.5.1 One-to-One Communication
  - 1.5.2 One to many Communication
    - 1.5.2.1 To a list
    - 1.5.2.2 To a group (channel)
    - 1.5.2.3 To a host/server mask
  - 1.5.3 One to all
    - 1.5.3.1 Client to Client
    - 1.5.3.2 Clients to Server
    - 1.5.3.3 Server to Server
- 1.6 What is an applet ? Difference between applets and application.
- 1.7 Out line for rest of the report

## CHAPTER - 2

### BASIC CONCEPT FOR DESIGNING AN APPLET

- 2.1 Applet overview

- 2.2 The essential Applet methods.
- 2.3 Applet Parameters
- 2.4 Communication between the Applet and the Browser
- 2.5 Using the Threads in Applets.
- 2.6 Inter Applet communications within the Browser.
- 2.7 Graphics class concepts. in appletes
- 2.8 Class and interface necessary for developing an Applet.

### **CHAPTER - 3**

**55 - 68**

#### **THEORITICAL CONCEPT OF NETWORKING AND CLLIENT SERVER**

- 3.1 Client server applilcations
- 3.2 Connection oriented protocol
- 3.3 Connectionless protocol
- 3.4. Sockets
- 3.5 Classes and Interface for developing netwrok application using java.

### **CHAPTER - 4**

**68 - 111**

- 4.1 Detail Analysis Design and implementation for Client Server information
  - 4.11. Building applet
  - 4.12. How the applet works
- 4.2. Detail Analysis Design and implementation for Chat Sever protocol.,
  - 4.2.1 Understanding Chat Areas
  - 4.2.2 Creating Our Own Chat Prtocol
  - 4.2.3 Building a Chat Applet
  - 4.2.4 Handling the Chat Applet
  - 4.2.5 Processing Messages Recived From a Chat Server.
  - 4.2.6 How a Chat Server Accept Clients.
  - 4.2.7 Creating a Chat Server's client Thread.
  - 4.2.8 Implementing Chat Server Methods.

### **CHAPTER 5**

**112 -113**

- 5.1 Conclusion.
- 5.2 References

## **FIGURES WITHIN THIS PROJECT REPORT**

- 1.1 Format Of IRC Server Network
- 1.2 Sample Small IRC Network
- 2.1 The Applet Class Hierarchy
- 2.2 Sequence Of Init Start, Stop And Destroy Class For An Applet
- 2.3 Pathways Of Data Exchange Between The Applet, Applet Context And Applet Stub Object
- 2.4 Class Diagram For The Graphics Class
- 2.5 Class Diagram Of The Applet Class
- 3.1 A Client Server Communication Scenario
- 3.2 Distribution Of Hosts Among Different Subnets Over The Internet
- 3.3. Conceptual Representation Of Internet Addressing
- 3.4 Class Diagram For Inet Address Class.
- 3.5 Class Diagram For Server Socket Class.
- 4.1 Sequence of Actions in the Client-Server Information

# ABSTRACT

Everybody uses e-mail, and most Internet users have been on Usenet news groups or visited the forums in CompuServe or AOL. But despite the strengths of these technologies, they share one major limitation: They're not real-time. E-mail and Usenet are messaging systems, and although they can approximate the feel of conversation at times, they lack the true spontaneity of live interaction. In some ways, that can be an advantage - people can compose and read messages as their schedules allow - but we all know that sitting around a table talking with one another has a lot of advantages too. So in this report I have developed real time interactive and dynamic Applet so that a group of people can talk by typing message. Also I have developed client server information applet to get the client server information which is used in chat server.

# CHAPTER 1

## INTRODUCTION

Chat means basically we spend our time in one or more windows, each representing a different channel or user. The window is split into two panes : the viewing area and the composing area. We read the incoming messages in the viewing area, and we type our own messages in the composing area. . What we type doesn't appear until you press the Enter key or click the Send button, so you have a chance to edit what you'll send. In practice, however, users dash off comment, question, or reply, then send it quickly for the sake of staying with the conversation.

Running a Chat server on an existing intranet or extranet server allows companies to offer live interaction among employees and associates, and public chat areas let organizations host discussions across the Internet.

### **Internet Relay Chat Protocol**

The IRC protocol was developed over the last 4 years since it was first implemented as a means for users on a BBS to chat amongst themselves. Now it supports a world-wide network of servers and clients, and is stringing to cope with growth. Over the past 2 years, the average number of users connected to the main IRC network has grown by a factor of 10.



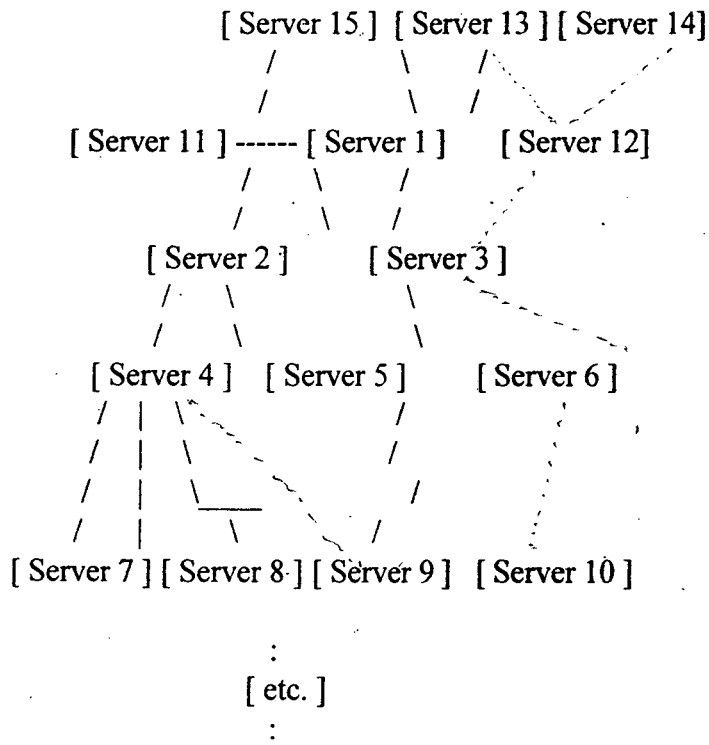
The IRC protocol is a text-based protocol, with the simplest client being any socket program capable of connecting to the server. The IRC (Internet Relay Chat) protocol has been designed over a number of years for use with text based conferencing.

The IRC protocol has been developed on systems using the TCP/IP network protocol, although there is no requirement that this remain the only sphere in which it operates.

IRC itself is a teleconferencing system, which (through the use of the client-server model) is well-suited to running on many machines in a distributed fashion. A typical setup involves a single process (the server) forming a central point for clients (or other servers) to connect to, performing the required message delivery/multiplexing and other functions.

## 1.1 Servers

The server forms the backbone of IRC, providing a point to which clients may connect to to talk to each other, and a point for other servers to connect to, forming an IRC network. The only network configuration allowed for IRC servers is that of a spanning tree [see Fig 1.1] where each server acts as a central node for the rest of the net it sees.



[ Fig.1.1. Format of IRC server network ]

## 1.2 Clients

A client is anything connecting to a server that is not another server. Each client is distinguished from other clients by a unique nickname having a maximum length of nine (9) characters. See the protocol grammar rules for what may and may not be used in a nickname. In addition to the nickname, all servers must have the following information about all clients: the real name of the host that the client is running on, the username of the client on that host, and the server to which the client is connected.

### 1.2.1 Operators

To allow a reasonable amount of order to be kept within the IRC network, a special class of clients (operators) is allowed to perform general maintenance functions

on the network. Although the powers granted to an operator can be considered as 'dangerous', they are nonetheless required. Operators should be able to perform basic network tasks such as disconnecting and reconnecting servers as needed to prevent long-term use of bad network routing. In recognition of this need, the protocol discussed herein provides for operators only to be able to perform such functions.

A more controversial power of operators is the ability to remove user from the connected network by 'force', i.e. operators are able to close the connection between any client and server. The justification for this is delicate since its abuse is both destructive and annoying.

### **1.3 Channels**

A channel is a named group of one or more clients which will all receive messages addressed to that channel. The channel is created implicitly when the first client joins it, and the channel ceases to exist when the last client leaves it. While channel exists, an client can reference the channel using the name of the channel.

Channels names are strings (beginning with a '&' or '#' character) of length up to 200 characters. Apart from the the requirement that the first character being either '&' or '#'; the only restriction on a channel name is that it may not contain any spaces (' '), a control G (^G or ASCII 7), or a comma (',' which is used as a list item\ separator by the protocol).

There are two types of channels allowed by this protocol. One is a distributed channel which is known to all the servers that are connected to the network. These

channels are marked by the first character being a only clients on the server where it exists may join it. These are distinguished by a leading '&' character. On top of these two types, there are the various channel modes available to alter the characteristics of individual channels.

To create a new channel or become part of an existing channel, a user is required to JOIN the channel. If the channel doesn't exist prior to joining, the channel is created and the creating user becomes a channel operator. If the channel already exists, whether or not your request to JOIN that channel is honoured depends on the current modes of the channel. For example, if the channel is invite-only, (+I), then you may only join if invited. As part of the protocol, a user may be a part of several channels at once, but a limit of ten (10) channels is recommended as being ample for both experienced and novice users. If the IRC network becomes disjoint because of a split between two servers, the channel on each side is only composed of those clients which are connected to servers on the respective sides of the split, possibly ceasing to exist on one side of the split. When the split is healed, the connecting servers announce to each other who they think is in each channel and the mode of that channel. If the channel exists on both sides, the JOINS and MODEs are interpreted in an inclusive manner so that both sides of the new connection will agree about which clients are in the channel and what modes the channel has.

### **1.3.1 Channel Operators**

The channel operator (also referred to as a "chop" or "chanop") on a given channel is considered to 'own' that channel. In recognition of this status, channel operators are endowed with certain powers which enable them to keep control and some

sort of sanity in their channel. As an owner of a channel, a channel operator is not required to have reasons for their actions, although if their actions are generally antisocial or otherwise abusive, it might be reasonable to ask an IRC operator to intervene, or for the users just leave and go elsewhere and form their own channel.

**The commands which may only be used by channel operators are:**

KICK - Eject a client from the channel

MODE - Change the channel's mode

INVITE - Invite a client to an invite-only channel (mode +i)

TOPIC - Change the channel topic in a mode +t channel

A channel operator is identified by the '@' symbol next to their nickname whenever it is associated with a channel (ie replies to the NAMES, WHO and WHOIS commands).

## **1.4 The IRC Specification**

### **1.4.1 Overview**

The protocol as described herein is for use both with server to server and client to server connections. There are, however, more restrictions on client connections (which are considered to be untrustworthy) than on server connections.

### **1.4.2 Character codes**

No specific character set is specified. The protocol is based on a set of codes which are composed of eight (8) bits, making up an octet. Each message may be composed of any number of these octets; however, some octet values are used for control codes which act as message delimiters.

Regardless of being an 8-bit protocol, the delimiters and keywords are such that protocol is mostly usable from USASCII terminal and a telnet connection.

Because of IRC's scandinavian origin, the characters {}| are considered to be the lower case equivalents of the characters []\, respectively. This is a critical issue when determining the equivalence of two nicknames.

### **1.4.3 Messages**

Servers and clients send eachother messages which may or may not generate a reply. If the message contains a valid command, as described in later sections, the client should

expect a reply as specified but it is not advised to wait forever for the reply; client to server and server to server communication is essentially asynchronous in nature.

Each IRC message may consist of up to three main parts: the prefix (optional), the command, and the command parameters (of which there may be up to 15). The prefix, command, and all parameters are separated by one (or more) ASCII space character(s) (0x20).

The presence of a prefix is indicated with a single leading ASCII colon character (':', 0x3b), which must be the first character of the message itself. There must be no gap (whitespace) between the colon and the prefix. The prefix is used by servers to indicate the true origin of the message. If the prefix is missing from the message, it is assumed to have originated from the connection from which it was received. Clients should not use prefix when sending a message from themselves; if they use a prefix, the only valid prefix is the registered nickname associated with the client. If the source identified by the prefix cannot be found from the server's internal database, or if the source is registered from a different link than from which the message arrived, the server must ignore the message silently.

The command must either be a valid IRC command or a three (3) digit number represented in ASCII text.

IRC messages are always lines of characters terminated with a CR-LF (Carriage Return - Line Feed) pair, and these messages shall not exceed 512 characters in length, counting all characters including the trailing CR-LF. Thus, there are 510 characters maximum allowed for the command and its parameters. There is no provision for continuation message lines.

#### 1.4.3.1 Message format in 'pseudo' BNF

The protocol messages must be extracted from the contiguous stream of octets. The current solution is to designate two characters, CR and mLF, as message separators. Empty messages are silently ignored, which permits use of the sequence CR-LF between messages without extra problems.

The extracted message is parsed into the components <prefix>, <command> and list of parameters matched either by <middle> or <trailing> components.

**The BNF representation for this is:**

<message> ::= [ ':' <prefix> <SPACE> ] <command> <params> <CrLf>

<prefix> ::= <servername> | <nick> [ '!' <user> ] [ '@' <host> ]

<command> ::= <letter> { <letter> } | <number> <number> <number>

<SPACE> ::= ' '{ ' }

<params> ::= <SPACE> [ ':' <trailing> | <middle> <params> ]

<middle> ::= <Any \*non-empty\* sequence of octets not including SPACE



or NUL or CR or LF, the first of which may not be ':>

<trailing> ::= <Any, possibly \*empty\*, sequence of octets not including  
NUL or CR or LF>

<crlf> ::= CR LF

#### NOTES:

- 1) <SPACE> is consists only of SPACE character(s) (0x20). Specially notice that TABULATION, and all other control characters are considered NON-WHITE-SPACE.
- 2) After extracting the parameter list, all parameters are equal, whether matched by <middle> or <trailing>. <Trailing> is just a syntactic trick to allow SPACE within parameter.
- 3) The fact that CR and LF cannot appear in parameter strings is just artifact of the message framing. This might change later.
- 4) The NUL character is not special in message framing, and basically could end up inside a parameter, but as it would cause extra complexities in normal C string handling. Therefore NUL is not allowed within messages.
- 5) The last parameter may be an empty string.
- 6) Use of the extended prefix ([!! <user> ] ['@' <host> ]) must not be used in server to server communications and is only intended for server to client messages in order to

provide clients with more useful information about who a message is from without the need for additional queries.

Most protocol messages specify additional semantics and syntax for the extracted parameter strings dictated by their position in the list. For example, many server commands will assume that the first parameter after the command is the list of targets, which can be described with:

<target> ::= <to> [ "," <target> ]

<to> ::= <channel> | <user> '@' <servername> | <nick> | <mask>

<channel> ::= ('#' | '&') <chstring>

<servername> ::= <host>

<nick> ::= <letter> { <letter> | <number> | <special> }

<mask> ::= ('#' | '\$') <chstring>

<chstring> ::= <any 8bit code except SPACE, BELL, NUL, CR, LF and  
comma (',)>

Other parameter syntaxes are:

<user> ::= <nonwhite> { <nonwhite> }

<letter> ::= 'a' ... 'z' | 'A' ... 'Z'

<number> ::= '0' ... '9'

<special> ::= '-' | '[' | ']' | '^' | '\_' | '{' | '}'

<nonwhite> ::= <any 8bit code except SPACE (0x20), NUL (0x0), CR  
(0xd), and LF (0xa)>

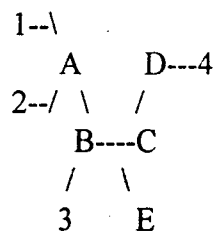
#### 1.4.4 Numeric replies

Most of the messages sent to the server generate a reply of some sort. The most common reply is the numeric reply, used for both errors and normal replies. The numeric reply must be sent as one message consisting of the sender prefix, the three digit numeric, and the target of the reply. A numeric reply is not allowed to originate from a client; any such messages received by a server are silently

dropped. In all other respects, a numeric reply is just like a normal message, except that the keyword is made up of 3 numeric digits rather than a string of letters.

#### 1.5. IRC Concepts.

This section is devoted to describing the actual concepts behind the organization of the IRC protocol and how the current implementations deliver different classes of messages.



Servers: A, B, C, D, E      Clients: 1, 2, 3, 4

[ Fig.1. 2 Sample small IRC network ]

### **1.5.1 One-to-one communication**

Communication on a one-to-one basis is usually only performed by clients, since most server-server traffic is not a result of servers talking only to each other. To provide a secure means for clients to talk to each other, it is required that all servers be able to send a message in exactly one direction along the spanning tree in order to reach any client. The path of a message being delivered is the shortest path between any two points on the spanning tree. The following examples all refer to Figure 1.2 above.

#### **Example 1:**

A message between clients 1 and 2 is only seen by server A, which sends it straight to client 2.

#### **Example 2:**

A message between clients 1 and 3 is seen by servers A & B, and client 3. No other clients or servers are allowed see the message.

#### **Example 3:**

A message between clients 2 and 4 is seen by servers A, B, C & D and client 4 only.

### **1.5.2 One-to-many**

The main goal of IRC is to provide a forum which allows easy and efficient conferencing (one to many conversations). IRC offers several means to achieve this, each serving its own purpose.

#### **1.5.2.1 To a list**

The least efficient style of one-to-many conversation is through clients talking to a 'list' of users. How this is done is almost self explanatory: the client gives a list of destinations to which the message is to be delivered and the server breaks it up and dispatches a separate copy of the message to each given destination. This isn't as efficient as using a group since the destination list is broken up and the dispatch sent without checking to make sure duplicates aren't sent down each path.

#### **1.5.2.2 To a group (channel)**

In IRC the channel has a role equivalent to that of the multicast group; their existence is dynamic (coming and going as people join and leave channels) and the actual conversation carried out on a channel is only sent to servers which are supporting users on a given channel. If there are multiple users on a server in the same channel, the message text is sent only once to that server and then sent to each client on the channel. This action is then repeated for each client-server combination until the original message has fanned out and reached each member of the channel.

The following examples all refer to Figure 1.2

**Example 4:**

Any channel with 1 client in it. Messages to the channel go to the server and then nowhere else.

**Example 5:**

2 clients in a channel. All messages traverse a path as if they were private messages between the two clients outside a channel.

**Example 6:**

Clients 1, 2 and 3 in a channel. All messages to the channel are sent to all clients and only those servers which must be traversed by the message if it were a private message to a single client. If client 1 sends a message, it goes back to client 2 and then via server B to client 3.

**1.5.2.3 To a host/server mask**



TH-6665

To provide IRC operators with some mechanism to send messages to a large body of related users, host and server mask messages are provided. These messages are sent to users whose host or server information match that of the mask. The messages are only sent to locations where users are, in a fashion similar to that of channels.

### **1.5.3 One-to-all**

The one-to-all type of message is better described as a broadcast message, sent to all clients or servers or both. On a large network of users and servers, a single message can result in a lot of traffic being sent over the network in an effort to reach all of the desired destinations.

For some messages, there is no option but to broadcast it to all servers so that the state information held by each server is reasonably consistent between servers.

#### **1.5.3.1 Client-to-Client**

There is no class of message which, from a single message, results in a message being sent to every other client.

#### **1.5.3.2 Client-to-Server**

Most of the commands which result in a change of state information (such as channel membership, channel mode, user status, etc) must be sent to all servers by default, and this distribution may not be changed by the client.

#### **1.5.3.3 Server-to-Server.**

While most messages between servers are distributed to all 'other' servers, this is only required for any message that affects either a user, channel or server. Since these are the basic items found in IRC, nearly all messages originating from a server are broadcast to all other connected servers.

## 1.6 WHAT IS AN APPLLET ? DIFFERENCE BETWEEN JAVA APPLLET AND APPLICATION

Java programs come in two flavours: applet and application simply speaking a java applet is a program that appears embedded in a web document; Java application is the term applied to all other kinds of java programs , such as those found in network servers and consmer electronics.

Traditionally, the word applet has come to mean any small application . In java , an applet is any Java program that is lunched from a Web document ; that is from an HTML file. Java applications, on the other hand, are programs that run from a command line, independent if a Web browser. There is no limit to the size or complexity of a Java applet. Java applets are in some ways more powerful than java applications. However, with the Internet where communication speed is limited and downlodad times are long, ,most Java applets ae necessarily small.

The technical differences between applets and applications stem from the context in which they run. A java applications runs in the simplest possible environment-its only input from the outside world is a list of command-line parameters. On the other hand, a Java applets receives a lot of information from the Web browser:it needs to know when it is initialized, when and where to draw itself in the browser window, and when it is activated or deactivated. As a consequence of these two very different execution environments, applets and applications have different minimum requirements.

The decision to write a program as an applet versus an application depends on the context of the program and its delivery mechanism. Because Java applets are always



presented in the context of a Web browser's graphical user interface, Java applications are preferred over applets when graphical displays are unnecessary. For example, A Hypertext Transfer Protocol(HTTP) server written in Java needs no graphical display; it requires only file and network access.

### **Differences between Java Applets and Applications**

	<b>Java Application</b>	<b>Java Applet</b>
Uses graphics	optional	Inherently graphical
Memory requirements	Minimal Java Application requirements	Java application requirements plus Web browser requirements
Distribution	Loaded from the file system or by a custom class loading process	Linked via HTML and transported via HTTP
Environmental input	Command line parameters	Browser client location and size; parameters embedded in the host HTML document
Method expected by the Virtual Machine	main-startup method	init-initialization method start-startup method stop-pause/deactive method destroy-termination method paint-drawing method

#### **1.7 Out line for Rest of the report**

In the Chapter 2 describes for procedure for designing an applet like .

1. essential applet method
2. applet parameter

3. using the threads in applets
4. communication between the applet and browser

**Chapter 3** describes the theoretical concept of networking, client server socket, internet address.

**Chapter 4** describes the Detail analysis,Design implementation in Java for client server information and chat server protocol applet.

Chapter -5 is for conclusion and reference

# CHAPTER - 2

## BASIC CONCEPT FOR DESIGNING AN APPLLET

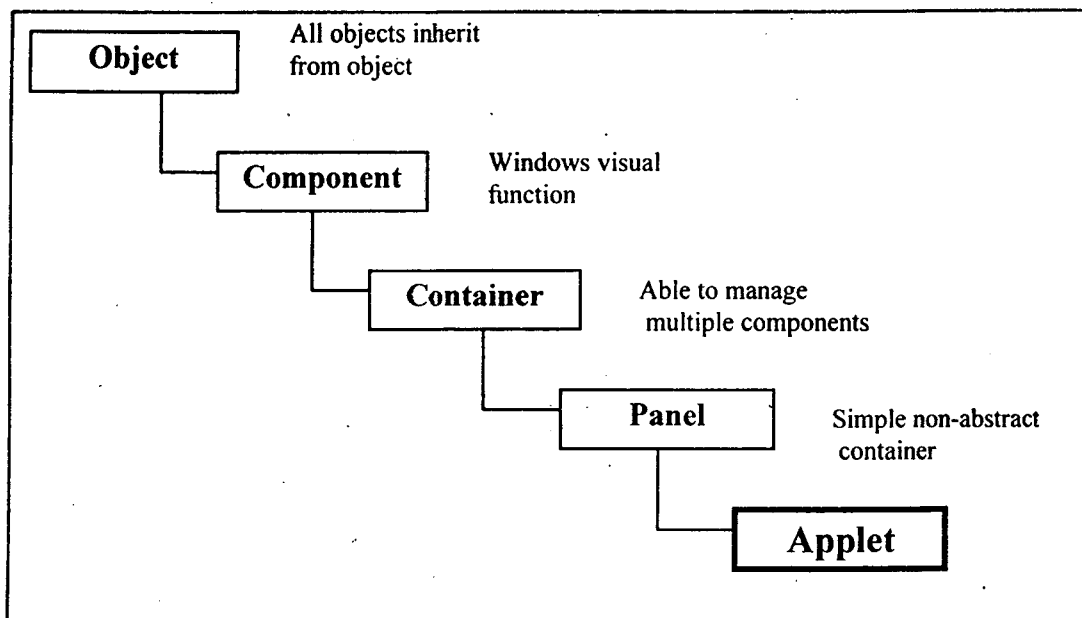
### 2.1 Applet Overview

Packaging interactive content in small, easily distributed objects was a design feature that had high priority to the developers of Java. To meet this design goal, they created the Applet class, along with several objects and interfaces designed to simplify image and audio processing.

An Applet is a custom interface component object, similar in concept to a Windows custom control, or an X-Windows widget. Applet-aware applications (or "applet browsers") can load and construct Applet objects from URLs pointing to Class files anywhere on a network, including the Internet, the largest network of them all. The Java Developer Kit's (JDK) Hot Java World Wide Web browser is an example of an applet-aware application. Using it, we can access interactive Applets from anywhere on the Internet, or Applets developed on the local file system. Security features of the Java language ensure distributed applets cannot damage or compromise the security of a local system.

Using the graphical capabilities of Java, applets are visually exciting multimedia elements. Through objects of the class `java.awt.Graphics` applets can create graphical on-screen content. The graphics class is included in this chapter because of the need for applets to display exciting visuals.

Because of all these features, applets have become the preferred method for distributing interactive content on the World Wide Web. A library of reusable, extensible Applets is one of the cornerstones of an Internet content creator's tool kit.



**Figure 2.1 The Applet class hierarchy**

## Applets

The above Figure above illustrates the Applet class hierarchy. Most ancestors of Applet in this hierarchy are Abstract windows Tool kit (AWT) classes. Throughout them, the Applet class inherits windowing capabilities. Specifically, the Applets display, surface drawing, and mouse and keyboard event handling functionalities are gained through these ancestors. All examples and discussions in this chapter stop short of utilizing AWT methods other than those that provides applets with their graphical capabilities. But keep in mind the rich set of facilities the AWT classes have when designing your own custom Applet classes.

Applet objects are created and controlled by a container application called an *Applet browser*. The applet browser arranges applet objects visually on the screen and dedicates a rectangle of screen space for the applet to display itself. Most applet browsers can manage more than a single Applet object at a time, and actually provide and interface for a Applet instance to communicate with each other.

## 2.2 The Essential Applet Methods

The actions of a custom Applet object are ruled by four essential methods : `Applet.init`, `Applet.start`, `Applet.stop`, and `Applet.destory`. The browser itself invokes these methods at specific points during the applet's lifetime. The `java.applet.applet` calls declares these methods and provides default implementations for them. The default implementations do nothing. Custom applets override one or more of these methods to perform specific tasks during the lifetime of the custom Applet object. Table following four methods, details when each is called by the browser, and whose what a custom applet's overriding implementation should do.

### Descriptions of the essential applet methods

Method	Description
<b>init</b>	Called once and only once when the applet is first loaded. Custom implementations allocate resources that will be required thorough the lifetime of the Applet object.
<b>destroy</b>	Called once and only once just before the Applet object is to be destroyed. Custom implementations release allocated resources, especially Native resources, which were loaded during <code>init</code> or during the lifetime of the Applet object.
<b>start</b>	called each time the applet is displayed or brought into the user's view on-screen. Custom implementations begin active processing or create processing threads.
<b>stop</b>	Called each time the applet is removed from the user's view. Custom implementations end all active processing. Background processing threads should either be destroyed in <code>stop</code> , or put to sleep and destroyed in the <code>destroy</code> method.

The proper place to allocate objects or load data required by the applet throughout its lifetime is `init`. This method is called only once during the lifetime of the applet, right after the object is created by the browser. Most custom Applets allocate resources required through the lifetime of the Applet object in this method. Another very common operation performed during `init` is to resize the applet's on-screen display surface using the inherited method `component.resize`. Some browsers display applets correctly only if the applet calls `resize()` in `init()`. The component class is described.

When the applet drops from view, for example because it is scrolled off the screen in the browser or the user opens a different document in the browser, the applet's `stop` method is called. This is the proper time for a custom Applet to cease any processing.

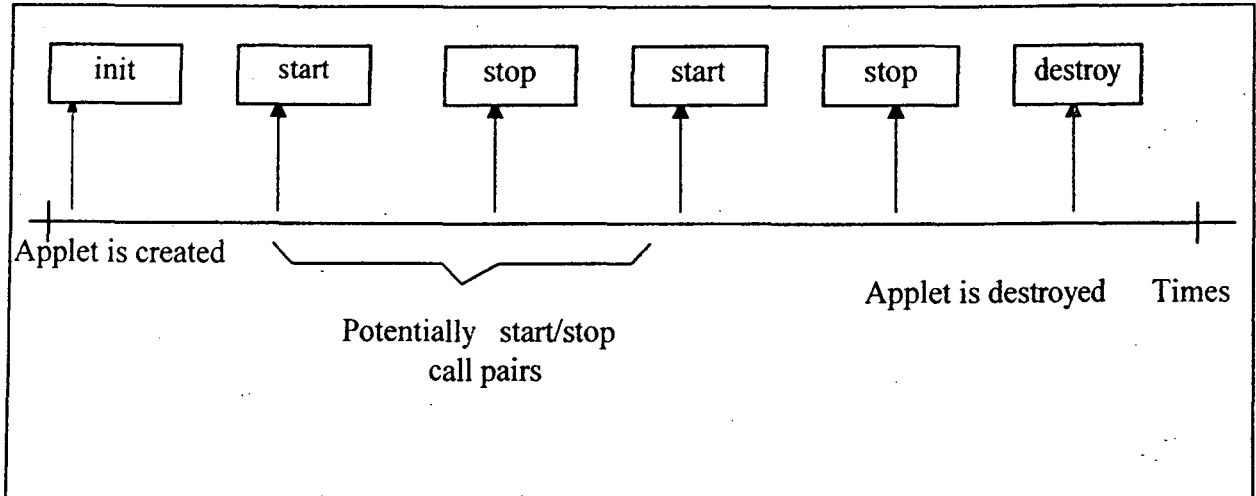
- Every call to `start` has a matching subsequent call to `stop`.
- The `start/stop`, for example if the applet is scrolled from the user's view and then scrolled back. When it is scrolled from the user's view, `stop` will be invoked. When it is scrolled back, `start` will be involved for the second time.

When the applet is finally and definitely to be unloaded from memory, `destroy()` is invoked. This is the appropriate time to delete any resources loaded during `init()`. The call to `destroy` is guaranteed to occur after the last call to `stop`. Note that while any resources allocated by an applet will automatically be cleaned up by Java's garbage collection facilities, it is more efficient to remove references to any allocated objects in `destroy`. Also note that resources allocated by "native" methods will not be cleaned up any the garbage collection facilities. Native resources must be explicitly released in `destroy`. (Native methods are platform-specific, dynamically loadable libraries accessible from within Java code. For the most part, Applet classes do not use native methods because of the severe security constraints placed on Applet objects.

### **2.3 Applet Parameters**

Similar to Java applications, applets can receive a process parameters. Applications receive parameters in the `argv[]` argument to the main method. The elements

of argv[] are the command line arguments to the application. Analogous to argv[], applet parameters are accessed within the applet code by the Applet. get Parameter method.



**Figure 2.2 Sequence of init, start, stop, and destroy calls for an Applet**

Conceptually, the browser maintains an internal listing of all the parameters passed on an embedded Applet object. The getParameter method accesses this internal list and retrieves the values specified for a uniquely named parameter. Our new listing uses the getParameter method to look up the value for the parameter named "Parameter name". If no such parameter was passed, getParameter would return null.

There is a method defined so that Applet so that Applet objects can publish a list of valid parameter names, valid values, and a description of getParameterInfo simply returns null, but an overriding implementation should return a String[n][3] 2-dimensional array where n is the number of unique parameters undertaken by members of the Applet class. Each row of three strings in this array should be of the format :

**{"parameter name", "valid value range", "text description"}**

There is no strict requirement on the format of any one of these strings. Each one should be suitable for textual display so that someone can read it. For example, the "valid

value range” string could be “0.-5, meaning the parameter s should be an integer between 0 and 5. This Applet class uses its Applet Context to access other active Applet instances. A detailed description of the Apple context interface and methods follows this discussion of Applet parameters.

Different types of browsers use different methods for passing parameters to applets. For example, applet-aware World Wide Web browsers generally use the HTML (APPLET> container tag to refer to applet code and parameter. Between the <APPLET> and </APPLET> name = [param-name] value = [param-val>]. No matter how parameters are passed into a particular browser, a loaded applet always uses getParameter to retrieve parameter values.

## **2.4. Communication Between the Applet and the Browser**

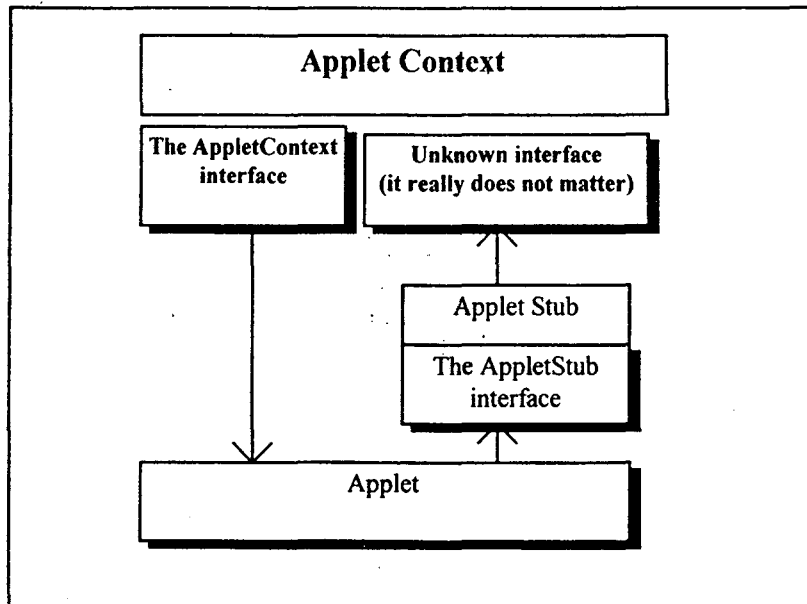
Applets obtain information about the state of the browser, what other Applet object are currently active, what is the current document opened by the browser, and so on, through the java.applet.Applet Context interface. The browser is abstracted by an object implementing this interface.

The browser also exposes some functionalities that an applet can use through this interface. For example, the loading of image and audio files into Java objects is handled transparently through the Applet context interface.

Between the Apllet Context and the Applet is an AppletStub object. Its purpose is to provide pathway for the exchange of applet-specific data between the AppletContext and the Applet. For example , the parameters for a specific Applet object are accessed by the Aplet through AppletStub.getParameter. AppletStub methods are translated into native or custom AppletContext method calls (the implementation of the completely up to the browser developers). An applet’s ApletStub is tightly wrapped by the java.applet.Applet implementation. So many so much so that all AappletStub functionalities are exposed as wrapper methods in the java.applet.Applet class. Therefore, a custom applet should never need to use its AppletStub directly. Figure 2.3



illustrates the pathways of data exchange between the Applet, the browser (abstracted by the AppletContext interface), and the AppletStub (the Applet's representative to the browser).



**Fig. 2.3 Path ways of data exchange between the Applet, Applet context and Applet stub object**

## 2.5 USING THREADS IN APPLETS

Much the same as applications, applets can create Threads to carry on background processing. A typical use of this would be an animation applet. To perform animation, the applet creates a new thread and starts it running in start. The animation Thread acts as a timer. Every so often, it wakes and draws a new frame in the animation sequence, then suspends itself until the next frame is to be drawn. In the applet's stop method, the animation thread is shutdown. Two versions of this simple animation technique are described in greater detail in the section on the Graphics class and methods. The important point here is that Threads generally are made to begin background processing an applet's start implementation and either suspended or destroyed in the applet's stop implementation.

We might assume that Threads created by an applet would be automatically halted by the browser when the applet is destroyed, so you wouldn't really need to suspend or

destroy a Thread object explicitly in stop. Instead, you could just leave it to Java's garbage collection facilities to destroy our Thread when the Applet object is destroyed. Many browsers, however, do not properly halt secondary applet threads, even after the applet has been destroyed, so the thread continues to execute after the applet has been destroyed. This is a result of applets relying on the Java garbage collection facility to destroy their threads. To ensure our custom applets behave as you want them to, include ceasing when you want them to cease, suspend any secondary threads in Applet.stop, and drop references to them in destroys.

## 2.6 INTER-APPLET COMMUNICATIONS WITHIN THE BROWSER

You can coordinate the activities of several applets by accessing and manipulating other Applet objects from within Applet code.

To obtain references to external Applets from within an applet you use the AppletContext getApplet and getApplets methods. The AppletNames Applet demonstrates this technique. Once a reference to another Applet is retrieved, your applet code can access any public member variable or method of the external Applet object. This code snippet retrieves an applet named "MyApplet" and calls one of its custom methods.

```
Applet applet = getAppletContext().getApplet("MyApplet");
if ( ! (applet instanceof MyAppletClass)) return;
MyAppletClass myapplet = (MyAppletClass)applet;

myapplet.CustomFunc()
```

GetApplet takes an applet "name" and returns a reference to the associated Applet object. This usage model implies the browser internally stores a unique String name associated with each applet, which can be used to look up the Applet in the internal browser storage.

## **GRAPHICS**

Applets are capable of displaying exciting and complex graphics and multimedia visuals. All graphical drawing operations in Java are performed through objects derived from the Graphics class. Whether you are drawing images downloaded from the Internet, drawing graphical primitives such as rectangles and arcs, or rendering text, all graphical operations are done using a Graphics class instance.

### **2.7 The Graphics Class Concept in Applet**

Each Graphics object is associated with two-dimensional “drawing surface,” analogous to the piece of paper on the drafting table. For example, the drawing surface can be a rectangle of a user’s on-screen desktop, as is the case when dealing with Applets or Windows. Other drawing surface types could also be associated with a Graphics object. The drawing source could be a binary image, stored in memory and never directly displays to the user. It could also be a page in a printer, or fax machine, or even a PostScript or other graphics-format file stored on a disk.

The “tools” of a Graphics object, the methods of the Graphics class are to draw onto the associated drawing surface. Rectangles, ovals, arcs, polygons, lines, text, and images can also be drawn onto the drawing surface using the various Graphics class methods.

The internal state of a Graphics object can be described by eight state variable, which can be modified using Graphics class methods.

- The foreground color
- The background color
- The current font
- The painting mode
- The origin of the Graphics object
- The horizontal and vertical scaling factors
- The “clipping” rectangle

- The drawing surface the Graphics object has been associated with

### **The Coordinate System of the Drawing Surface**

All drawing surfaces use the same two-dimensional coordinate system. The X axis is in the horizontal direction of the drawing surface, and increases from left to right on the drawing surface. The Y axis is in the vertical direction, and increases from top to bottom.

The Graphics object origin defines where its X and Y axes cross, and is identified by the point (0,0). A scaling factor is assigned to both axes, which defines how quickly the coordinated increase along with axis. By default, when the Graphics object is first created, the origin lies in the upper-left corner of the drawing surface, and the scaling factor along both axes is one.

The Graphics object's X and Y axes stretch to what is essentially an infinite distance in all four directions. However, only coordinates within the Graphics object "clipping rectangle" are of any interest. That's because graphical operations cannot be performed outside this rectangle. Such operations will not result in any sort of error, but neither will they have any effect on the drawing surface.

The clipping rectangle of a Graphics object represents the physical boundaries of the associated drawing surface. For example, a Graphics object associated with a 100 pixel by 100 and a height of 100. For on-screen desktop and in-memory image drawing surfaces, each Graphics coordinate represents a single pixel of the drawing surface. Hence, a 100 pixel by 100 pixel rectangle is represented by a 100 by 100 clipping rectangle in the associated Graphics object.

### **Obtaining Graphics Objects**

A program cannot create its Own Graphics objects, but instead must ask the Java runtime system to create them for specific display surfaces. Without using custom classes implementing native methods, only two types of display surfaces can be accessed through Graphics objects.

- Sections of the on-screen desktop surface are accessed through Graphics objects passed to the update and paint methods.
- In-memory Image objects are accessed through Graphics objects created by Image create Graphics.

Applets inherit the update and paint methods from the Component class, which the Applet class extends. Both of these methods are called automatically by the Java runtime system when it is time to display information to the user on the desktop. This code snippet shows how a custom applet would override the default implementation of paint to control its display surface:

```
Public void paint(Graphics g) {  
    // Draw on the display surface here  
}  
}...
```

A graphics object is automatically created by the Java runtime system and passed to paint. This Graphics object has a clipping rectangle set to the exact dimension of the Applet's display surface. In the case where only a portion of the Applet must be redrawn, such as when another window temporarily covers part of the Applet's display surface, the dimensions may be smaller.

The only other method for obtaining a Graphics object is using Image.createGraphics. An applet or application calls this method directly. The Graphics object that is returned is capable of rendering geometric primitives, text, and other Image object onto the Image. This is useful for the so-called "double-buffered" drawing technique, used widely to effect a smooth transition between animation frames. You'll learn more about this technique in the upcoming discussion of animation.

## **The Geometric Primitives**

All Graphics objects are able to render several different types of geometric primitive drawing objects on a drawing surface.

### **Geometric Primitives**

<b>Primitive</b>	<b>Representation Through Rendering Methods</b>
<b>Rectangle</b>	The point of the upper-left corner of rectangle relative to the Graphics origin, the rectangle's width and height.
<b>Rounded rectangle</b>	The point of the upper-left corner of the rectangle relative to the Graphics origin, the rectangle's width and height
<b>3D rectangle</b>	The point of the upper-left corner of the rectangle relative to the Graphics origin, the rectangle's width, height, and the raising or depressing implication of the beveled edges.
<b>Oval</b>	A bounding rectangle defines the size and shape of the oval. This rectangle is described the same way as a rectangle geometric primitive.
<b>Arc</b>	An arc is a section, or pie edge, or an oval. an arc is described by the bounding rectangle of an oval, the starting angle of the arc, and the angular length of the arc.
<b>Polygon</b>	An ordered a set of points defines the vertices of a polygon to Graphics rendering methods. Alternatively, a Polygon object can be used, through Polygons are essentially just on ordered set of vertices. Points are all relative to the Graphics object's origin.
<b>Line segment</b>	Two points defining the two end points of the line segment. Both points are relative to the Graphics object's origin.

All primitive can be rendered in either outlined or filled form, except the Line primitive, which cannot be filled. The outlined version of a primitive is rendered using the primitive's "draw" method. For example, Graphics.drawRect will render a rectangle as two sets of parallel lines using the Graphics objects current foreground color. The "fill" method is used to render a filled geometric primitive. Graphics.fillRect will render a solid rectangular block on the display surface using the current foreground color.

### The Painting Mode

The painting mode of a Graphics object is, default, set to "overwrite" mode. In this modern all graphics are rendered by overwriting the pixels of the display surface using the graphics object's current foreground color. We can force the Graphics object into overwrite mode using Graphics.setPaintMode. When called, this parameter less method places the Graphics into overwrite mode. Expressed pseudo-mathematically, the color of destination pixels after rendering is

$$\text{colorDest}(*x,y) = \text{graphics.foregroundColor}$$

The other method of modifying a Graphics object's painting mode is Graphics.setXORMode. When called, the Graphics object uses XOR mode for rendering geometric primitives, text, or Images on the drawing surface. Three colors are combined mathematically to determine the color of determine the color of destination pixels after rendering, as follows,

$$\text{colorAfterRendering}(x,y) = \text{colorBeforeRendering}(x,y) * \text{graphics.foregroundColor} * \text{graphics.alternateColor}$$

where the \* symbol represents a bit wise XOR operation. The alternate color of a Graphics object is specified as the only parameter to Graphics.stXORMode

## 2.8 Class and interface necessary for developing an Applet

The following classes and interface necessary for developing custom Applet object in Java.

	<b>Description</b>
<b>Class/Interface</b>	
<b>AppletContext</b>	Exposes services implemented by the applet browser for user by Applet objects. Conceptually, all active Applet object have access to the same AppletContext.
<b>Graphics</b>	Encapsulates a drawing surface, and exposes tools for drawing graphics and rendering text on that drawing surface. A drawing surface may be a rectangle of the desktop, on in-memory image, or even a page in the printer.
<b>Applet</b>	Represents on emendable Applet object.

### APPLET CONTEXT

**Purpose** An interface which abstracts the browser to an Applet. Methods for testing and modifying the current state of the browser are provided as public members of this interface.

**Syntax** interface AppletContext

**Description** Applet gets its AppletContext using Applet.getAppletContext. Using this interface, the Applet can get and set some parameters of the browser's current state. An Applet can get and set some parameters of the browser's current state. An Applet can get references to other



Applets currently running in the browser, download images and auto clips, and load a new document into the browser through the AppletContext interface.

<b>Package Name</b>	java.applet
<b>Imports</b>	java.awt.Image, java.awt.Graphics, java.awt.image.ColorModel, java.net.URL, java.util.enumeration
<b>Constructors</b>	None
<b>Parameters</b>	None

## **GETAPPLET**

<b>Interface</b>	AppletContext
<b>Purpose</b>	Used to facilitate inter-applet communications within a browser.
<b>Syntax</b>	public Applet getApplet (String srtName);
<b>Parameters</b>	None
<b>String SrtName</b>	This interface methods implies the browser stores, with each loaded applet, a unique string to identify that applet. It passes to getApplet one of these unique applet identifiers to gain access to the associated Applet object.
<b>Description</b>	Multiple Applet objects can be simultaneously loaded and run by the

same browser. Each applet runs within its own Thread. Use this method to access other applets running concurrently. It is completely up to a particular browser how to associate a particular string with an Applet object. For example, most commercial-grade World Wide Web browsers which are applet-aware use the NAME tag in the <APPLET> container tag to associate a name string with a particular applet, as in the HTML snippet below.

**Imports** None

**Returns** The Applet object associated with the unique String StrName. If no applet is associated with strName, null is returned or if the applet browser does not provide facilities for inter-applet communications.

## GETAPPLETS

**Interface** AppletContext

**Purpose** Used to facilitate inter-applet communications within a browser.

**Syntax** `public Enumeration getApplets();`

**Parameters** None.

**Description** This method allows you to look up all applets currently running in the browser. The browser which implements this method will give your access to all Applet objects currently running in the browser.

**Imports** None

**Returns** An Enumeration object is returned. Each element in the Enumeration's is an Applet currently active in the browser. Note that an empty Enumeration, or a return of null, could be interpreted in two ways : Either getApplets() is not fully implemented by the browser, or no other applets are active in the browser.

No exact specification currently exists describing what getApplets should return in either of these situations..

## **GETAUDIOCLIP**

**Interface** AppletContext

**Purpose** Loads an audio file and reads it to be played by the browser.

**Syntax** Public AudioClip getAudioClip(URL url);

**Parameters** Points to an audio file to be loaded by the browsers  
**(URL url)**

**Description** Commercial-grade browsers, especially World Wide Web browsers, have built-in facilities for loading and playing audio files. Appletes used the getAudioClip method to load audio files from any URL the browser can understand. Applets should use one of the overloaded Applet.getAudioClip methods to access AudioClips instead of AppletContext.getAudioClip. This method is rarely called by an Applet directly.

**Imports**            java.net.URL

**Returns**            The object returned by this function implements the AudioClip interface. If the URL is no understood by the browser, null will be returned or if the browser does not provide this functionality to applets.

## **GETIMAGE**

**Interface**        AppletContext

**Purpose**            To load an image from a URL and prepare it for rendering on a display surface.

**Syntax**           Public Image getImage(URL url);

**Parameters**      Points to an image file to be loaded by the browser.

**URL url**

**Description**      Java applications must implement methods for reading and interpreting image files, and converting the image data into Image objects. Applets may have this functionality exposed to them by the browser through the AppletContext.getImage method. Browsers that can load and interpret various image formats, such as GIF, JPEF or TIFF, can provide that capability to applets. Applets simply provide a URL pointing to an image file in a recognized format. No methods are provided for an applet to query which image formats are supported by a browser. Therefore, it is usually a good idea to only try to load images in very common graphics

formats, such as GIF or JPEG.

**Imports**        `java.awt.Image`

**Returns**        An Image object will be returned by this object, or null if this facility is not supported by the browser. The reaction of this methods when the URL reforest an unsupported protocol, or when the image file format is unrecognized, is unspecified. Generally, it can be assumed that null will be returned if these capability it not provided by the browser.

## **SHOWDOCUMENT**

**Interface**        `AppletContext`

**Purpose**            Opens a new document in the browser. An overloaded version exists to specify the name of the target browser frame.

**Syntax**            `public void showDocument (URL,url);`  
`public id showDocument (URL url, String target);`

**Parameters**      Points to the document to be opened by the browser. If the protocol referred to by the URL is not recognized by the browser, this call will be ignored. If the document format implied by the URL's field name is not recognized by the browser, this call will be ignored.

**URL url**

**Description**      In the abstract, Applets are seen as being embedded in distributed "documents," such as World Wide Web pages. When implemented, this method allows the applet to force the browser to open a particular document pointed to by a URL. Like all other methods in this

interface, a particular browser may not implement this method, in which case the browser will simply ignore a call to this method.

If the second overloaded version of this method is used, then the document will be opened in a browser frame with the same name as the target parameter.

**Imports**            java.net.URL

**Returns**            The Applet object associated with the unique Starting strName. If no applet is associated with strName, null is returned or if the applet browser does not provide facilities for inter-applet communications.

**Example**            This applet asks the browser to reload the current document whenever the Applet's stop method is invoked.

```
public class RestartingApplet extends Applet {  
    public void stop() {  
        AppletContext ac = getAppletContext();  
        if (null != ac)  
            ac.showDocument(getDocumentBase());  
    }  
}
```

## GRAPHICS

**Purpose**            An AWT Component (such as an Applet) uses a Graphics object to draw on display surface.

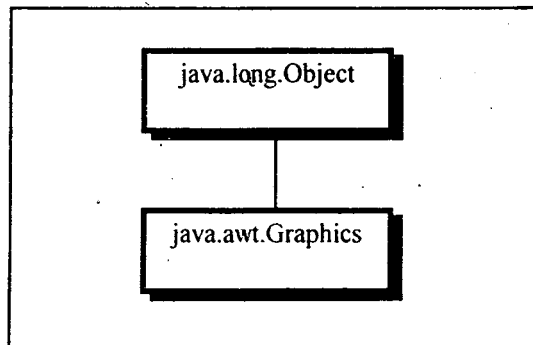
**Syntax**            public class Graphics

**Description**        A Graphics object is always associated with a "display surface." The display surface can be a rectangle of the on-screen desktop, an Image in memory, or potentially any rectangular area that can be drawn on. You use the Graphics class methods to render graphics and text on the display surface associated with the Graphics object. Figure 2.4 shows the class diagram for the

## Graphics class

**Package Name**      java.awt

**Imports**            java.awt \*, java.image.ImageObserver



**2.4 Class diagram for the Graphics Class**

### **CLEARRECT**

**Class Name**      Graphics

**Purpose**            To erase the specified rectangle using the background color of the display surface associated with the Graphics object.

**Syntax**            public abstract void clear Rect(int x, int y, int width, int height);

**Parameters**      These four parameters define the rectangle to be erased on the display surface.

**int x**

**int y**

**int width**

**int height**

**Description** This method is used to erase a rectangle from the display surface. The associated display surface's background color is used to fill the specified rectangle. This is a legacy method which was never removed from the alpha release of Java. Use of this method is not advised. Instead, use `Graphics.fillRect`, specifying the color you want to use to erase the rectangle. It is an unfortunate but true fact that the Java API does not specify an overloaded version of this method which takes `Rect` object as a parameter. The origin and extent of the rectangle must be explicitly provided in the four parameters to this method.

**Imports** None.

**Return** None

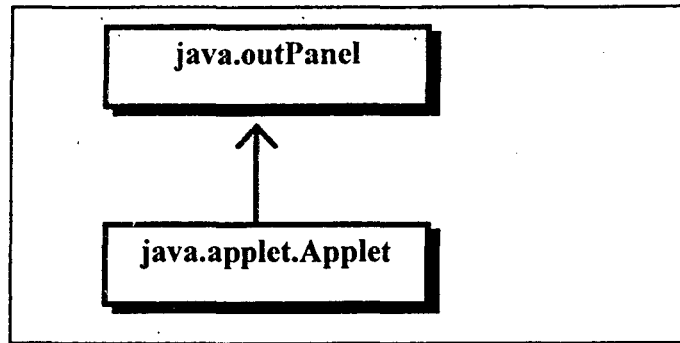
## **APPLET**

**Purpose** An embeded interactive component, suitable for embedded in World Wide Web pages using special HTML tags.

**Syntax** `public class Applet extends Panel`

**Description** A java Applet is an interactive Component special designed for use across the World Wide Web. The Applet class defines methods for controlling the lifetime of an Applet object, for which your applets provide custom implementations. Each applet running in an applet-aware browser has its own Thread, which uses the Applet methods `init`, `start`, `stop` and `destroy` to control the applet's lifetime. the Applet communicates with the browser through Applet context and Applet stub objects.





**Fig. 2.5 Class Diagram of The Applet Class**

### ISACTIVE

- Class Name** Applet
- Purpose** Indicates whether or not the Applet has been started.
- Syntax** public boolean isActive();
- Parameters** None.
- Description** Just before the before the Applet's start method is called, the Applet is marked as "active". At that point, shall calls to this method return true. Before that time and just before destroy is called, the Applet is marked as not active.

### GETDOCUMENTBASE

- Class Name** Applet
- Purpose** Gets the URL for the document this Applet is embedded in.
- Syntax** public URL getDocumentBase();
- Parameters** None.
- Description** The URL for the document this Applet is embedded in is returned. This method is shallow wrapper around AppletStub.getDocumentBase, so if the AppletStub is not implemented then, a call to this method will cause a NullPointerException to be throw.

## GETCODEBASE

<b>Class Name</b>	Applet
<b>Purpose</b>	Gets the URL for this Applet's CLASS file.
<b>Syntax</b>	public URL getCodeBase();
<b>Parameters</b>	None.
<b>Description</b>	The URL for the this Applet's CLASS file is returned. This method is a shallow wrapper around AppletStub.getCodeBase, so if the AppletStub is not implemented, then a call to this method will cause a NullPointerException to be thrown.

## GETPARAMETER

<b>Class Name</b>	Applet
<b>Purpose</b>	Gets the string value of a particular Applet parameter.
<b>Syntax</b>	public String getParameter (String name);
<b>Parameters</b> <i>String name</i>	Name of the parameter to retrieve. This is the value of the "name" tag within the HTML <PARAM> field which defines the Applet.
<b>Description</b>	This method returns one of the parameters to this Applet. Parameter are declared between the <APPLET> tag has two possible field : "name" and "value". By indicating one of the valid names for this Applet, the corresponding "value" field string will be returned.

## GETAPPLETCONTEXT

<b>Class Name</b>	Applet
<b>Purpose</b>	Retrieve the Appletconext for this Applet.
<b>Syntax</b>	public AppletContext getAppletContext();
<b>Parameters</b> <i>String name</i>	None

**Description** The AppletContext represents the browser this Applet is being displayed on. To retrieve a reference to an Applet's AppletContext, use this method.

## SHOWSTATUS

**Class Name** Applet

**Purpose** Displays a message on the browser's status bar.

**Syntax** public void showStatus(string msg);

### Parameters

**String msg** Message to be displayed on the browser's status bar.

**Description** Browsers generally have a status bar below the main display window. Use this method to place a message within that status bar. This method is an shallow wrapper around AppletContext.showStatus. If the Applet is not created within the context of a browser which implements AppletContext, then a call to this method will throw a NullPointerException.

## GETIMAGE

**Class Name** Applet

**Purpose** Creates an Image object from a URL pointing to a graphics-format file.

**Syntax** public Image getImage (URL url);  
public Image getImage(URL url, String str);

### Parameters

**URL url** URL of the graphics-format file containing the Image's data.

**Description** This method creates an Image object from a URL pointing to a graphics format file. The Image data is not downloaded until it is accessed at some point later in the Applet's execution. To force the Image to be loaded, use a Media Tracker object. The second overloaded version allows you to specify a base and relative URL to the graphics-format file. This method is a shallow wrapper around AppletContext.getImage.

## GETAUDIOCLIP

<b>Class Name</b>	Applet
<b>Purpose</b>	Creates an AudioClip object from a URL pointing to an audio data file.
<b>Syntax</b>	<code>public AudioClip getAudioClip(URL url);</code>
<b>Parameters</b>	<code>public AudioClip getAudioClip(URL url, String str);</code>
<b>URL url</b>	URL of the audio data file containing the AudioClip's data.
<b>Description</b>	This method creates a AudioClip object from a URL pointing to an audio data file. The AudioClip's data is not downloaded until it is accessed at some point later in the Applet's execution. The second overloaded version allows you to specify a base and relative URL to the audio data file. This method is a shallow wrapper around <code>AppletContext.getAudioClip</code> .

## GETAPPLETINFO

<b>Class Name</b>	Applet
<b>Purpose</b>	Custom implementations return a text String describing this Applet.
<b>Syntax</b>	<code>public String getAppletInfo();</code>
<b>Parameters</b>	None.
<b>Description</b>	Your custom applets should implement this method to return an information string about the applet. This string may include copyright information, or information about where to download the applet from, etc.

## GETPARAMETERINFO

<b>Class Name</b>	Applet
<b>Purpose</b>	Custom applets can expose text information about the parameters this applet understands by implementing this method.
<b>Syntax</b>	<code>public String[][] getParameterInfo();</code>

**Parameters** None

**Description** It is easy to make your applet self-describing by implementing this method. Have your implementation return a set of arrays of Strings. Each array of String should contain exactly three elements. The first String of each array is the name of a parameter the Applet understands. The second is a textual description of valid values for that parameter, such as "1-10" or "url". The third is a textual description of how the parameter is used, such as "URL for the background image".

## GETPARAMETERINFO

**Class Name** Applet

**Purpose** Downloads and plays an AudioClip from an audio data file.

**Syntax** public void play (URL url, String str);

### Parameters

**URL url** URL or base of a relative URL to the audio data file for the AudioClip and want to play.

**Description** Relative URL to the URL you want to play

This method is a simple shorthand for getting an AudioClip and playing it. Use of this method saves about three lines of explicit coding.

## INTO

**Class Name** Applet

**Purpose** Called by the Applet's Thread to start it running.

**Syntax** public void start it running.

**Parameters** public void start();  
None

### Description

The start() method is one of the four methods which define an Applet's action during its lifetime. In our custom applet, implement this method to actually perform the applet's behavior. The start() method is potentially called several times during the lifetime of the applet. Each call to start() is matched by exactly one subsequent call to stop(), sometime in the future.

A typical operation performed in the start() method is kick-starting the applet's background Threads.

## **START()**

**Class Name** Applet

**Purpose** Called by the Applet's Thread to start it running.

**Syntax** public void start();

**Parameters** None

**Description** The start() method is one of the four methods which define an Applet's action running its lifetime. In your custom applet, implement this method to actually perform the applet's behavior. The start() method is potentially called several times during the lifetime of the applet. Each call to start() is matched by exactly one subsequent call to stop(), sometime in the future. A typical operation performed in the start() method is kick-starting the applet's background Threads.

## **STOP**

**Class Name** Applet

**Purpose** Called by the Applet's thread to stop it running.

**Syntax** public void stop();

**Parameters** None

**Description** The stop() method is one of the four methods which define an Applet's action during its lifetime. In your custom applet, implement this method to gracefully shut down the applet. The stop() method is potentially called several times during the lifetime of the applet. Each call to stop() is match by exactly one prior call to start(). Stop any background Treads from processing before returning from you custom implementation of this method.

## **DESTROY()**

**Class Name** Applet

**Purpose** Called by the Applet's Thread to allow it to perform initial clean-up.

**Syntax** public void destroy();

**Parameters** None.

**Description** The destroy() method is one of the four methods which define an applet's action during its lifetime. In your custom applet, implement this method to deallocate any resources allocated during the applet's lifetime. The destroy() method is called exactly once, just before the Applet object is destroyed.

# CHAPTER -3

## THEORETICAL CONCEPT OF NETWORKING AND CLIENT SERVER

The computer network is a communication system for connecting two or more hosts. Hosts can be anything from microcomputers to super-computers, which makes establishing communication among them an involved task for programmers. The goal of java's internet working facilities is to hide the details of different physical networks from programmers. This allows the programmer not to worry about the more romantic pursuits of network programming and not to be bogged down by the trivial details of many different systems. However, this grand achievement of hiding details was no walk in the park for the java designers. Hosts can have vastly different physical attributes and may be dedicated to widely varying tasks. What is needed to make all these different species of system happy and able to communicate with each other is a common protocol. A protocol is a set of rules and conventions between the communicating participants. Using the higher-level protocol abstractions, the programmer can create Java programs quickly and with increased productivity. They need not build special version of application software to move and translate data between different type so machines.

This chapter introduces the basic concepts of networking. It discusses client-server applications, tells you how to identify a host using an Internet address, and explains what sockets are. The project I will develop in the chapter is a client-server application. The client sends messages to the server requesting it to send the contents of a



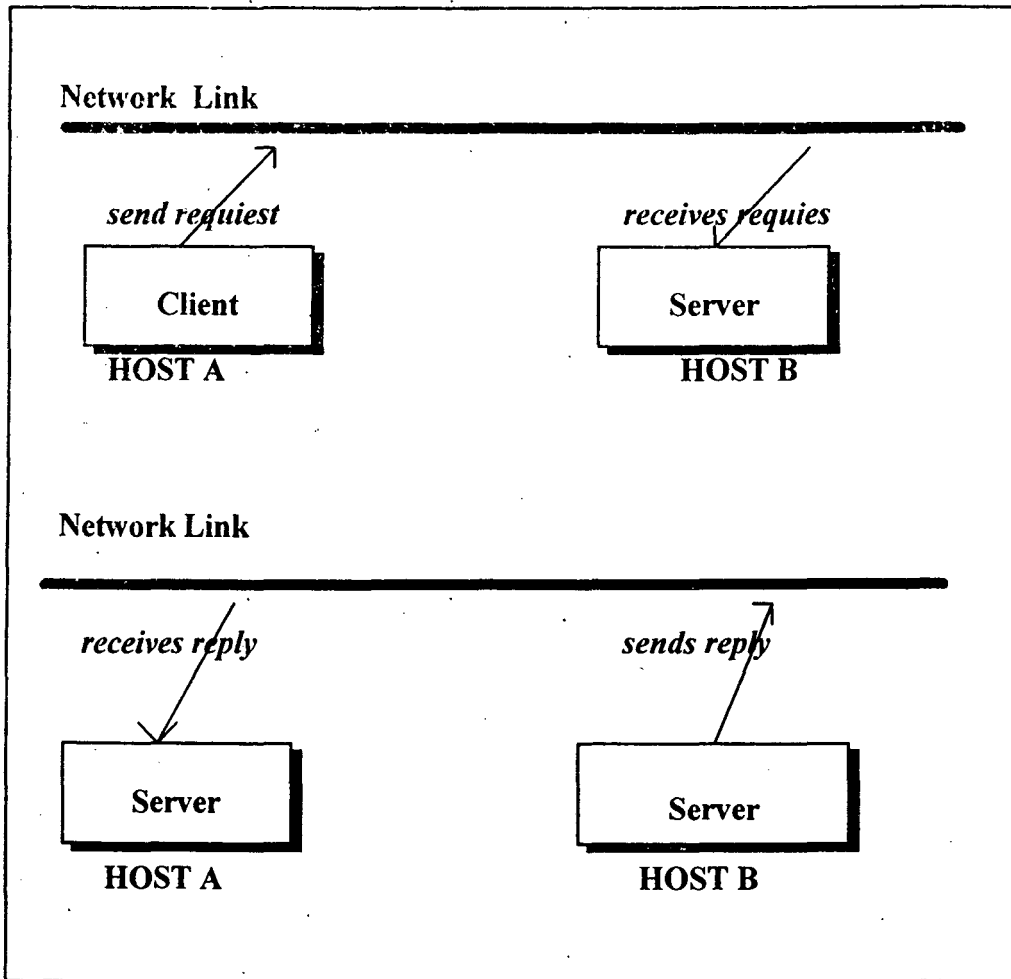
file. The server process the request and sends the contents of the file line-by-line. On receiving the file's contents from the server, the client displays it on a window.

### **3.1 Client-Server Applications**

There are several models for building network applications. The most widely used model is the client-server model which involves two types of processes: a server process and a client process. When you start a server process on a host, it waits for a client to contact it. A client process, started on the same host or a different one, sends a request to the server over the network. The server responds to the request by sending a reply.

The communication between a server and a client can be accomplished in two ways: connection-oriented or connectionless. In a connection-oriented transfer, a dedicated connection is established between a server and a client. They use this connection to exchange information. Given that the other type is called connectionless, it doesn't seem like a lot of communication actually happens between them. Then how do they communicate? The client sends the request by specifying the server's address. This is received by the server, who is waiting for a message from some client. The server obtains the client address from the message to which it may then respond.

- a) Host A, as a client sends a request for service to server located on host B
- (b) After processing the request, the server sends a reply to the client



**Fig. 3.1 A client-server communication scenario**

### 3.2 Connection-Oriented Protocol

In a connection-oriented communication, the client and the server have a dedicated link established between them. It is similar to the telephone communication system. When you call someone and the called phone number exist, there is a dedicated line for you to converse. Whatever you speak is guaranteed to be heard on the other side, which probably an element of delay. Also the words you speak are heard in the exact order in which they were spoken. Also connection-oriented protocol is a reliable protocol.

The messages sent between any two processes are guaranteed to be delivered and in the proper sequence. Most of the networking applications are connection oriented, as they require reliable communication protocol. TCP (Transfer Control Protocol) is a connection-oriented protocol in the TCP/IP family.

### **3.2 Connectionless Protocol**

In a connectionless protocol, there is no dedicated link between the client and the server. They send messages as datagram packets, each of which contains the destination address. The underlying network will targeted destination address from a packet and routes the packet to the destination. In this sense, each packet is self-contained. They have the information about the sender and the intended receiver, apart from hecore message. We can consider such a communication to be similar to the Indian Postal service. Each letter we send has its destination address contained in it and the postal department takes the necessary steps to route the mail to the destination. But we should note that the postal department guarantees neither the delivery not the sequence of delivery. Similarly, in the connectionless protocol, the packets are not guaranteed to be delivered, and even it they are delivered, the order of delivery is not guaranteed. Then the obvious question is : Where will you ever use the connectionless protocol for communication ? You can use this protocol where the order of messages is not critical. For example, consider a time server application. The server can keep sending the updated time to the client. The client need not assume any order of delivery. As it receives message, logic can be built in the client application to sense the sequence of received messages. In this application, missing

packets will not create any adhoc. In the TCP/IP family, the UDP is the connectionless protocol.

### 3.3 Internet Address

To identify a particular host in the Internet we need an Internet address. Using Internet addressing, a host can communicate with another host located in the same physical network or subnet, or in a different physical network, where both networks are linked by the Internet. Hence, hosts separated geographically can communicate effectively by addressing each other by their Internet. Hence, hosts separated geographically can communicate effectively by addresss in each other by their Internet address.

An Internet address is usually written as four decimal numbers, which are separated by decimal points. Each decimal digit in this address encodes one byte of the 32-bit Internet address. The Internet address maps to a unique host and the host can be addressed by a unique combination of the host and the name of the particular network the host is a part of (i.e., domain name).

In the Internet address representation, Internet ID identifies a network in the Internet. Subnet ID represents local area network(subnet) and Host Id is the identifier for a given host in the subnet. The combination of these three identifiers represents a unique host in the Internet. Here 128.230.32.66 is the Internet address that maps to ratnam.cat.syr.edu, where ratenam is the machine name and is a part of cat.syr.edu network domain. In this example, host Id is 66, subnet Id is 32 and the Internet Id is

128.230. The InetAddress Class in Java encapsulates the methods required to manipulate with an Internet address in a networking application.

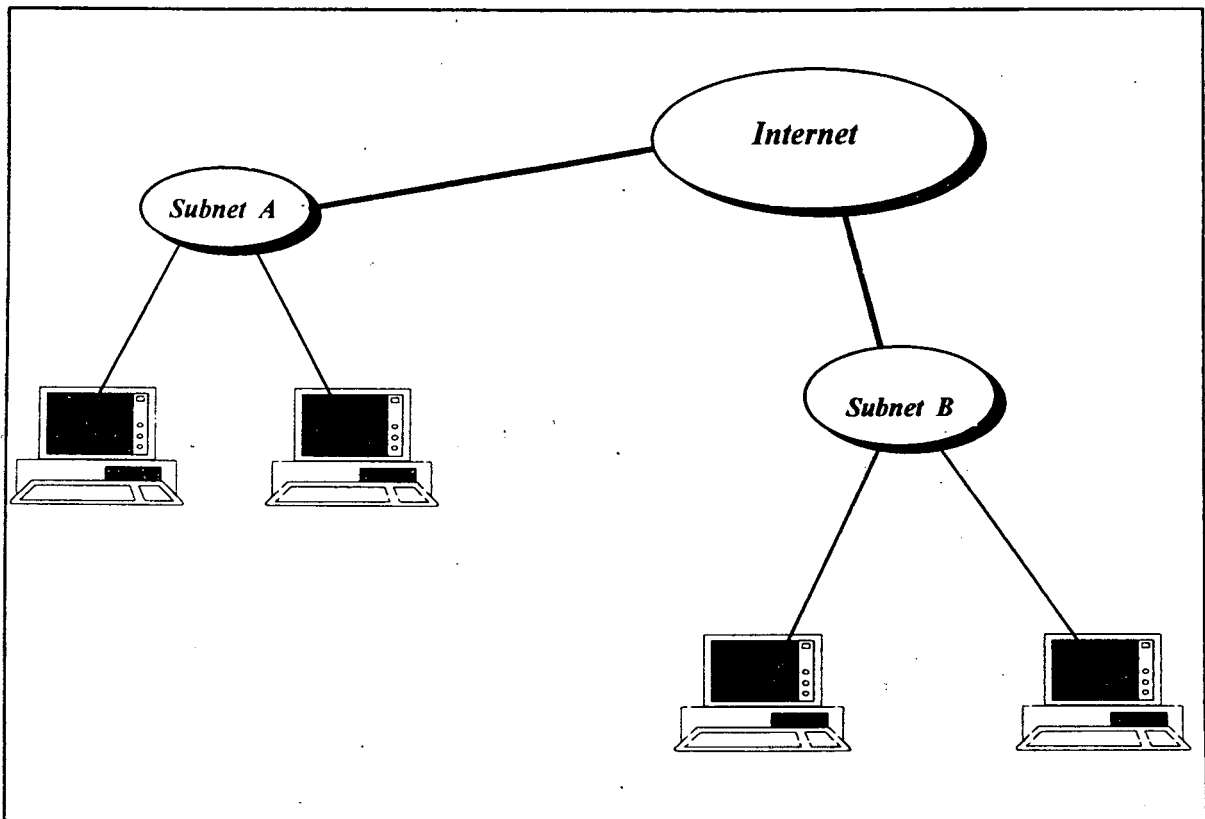


Fig. 3.3 *Distribution of hosts among different subnets over the Internet*

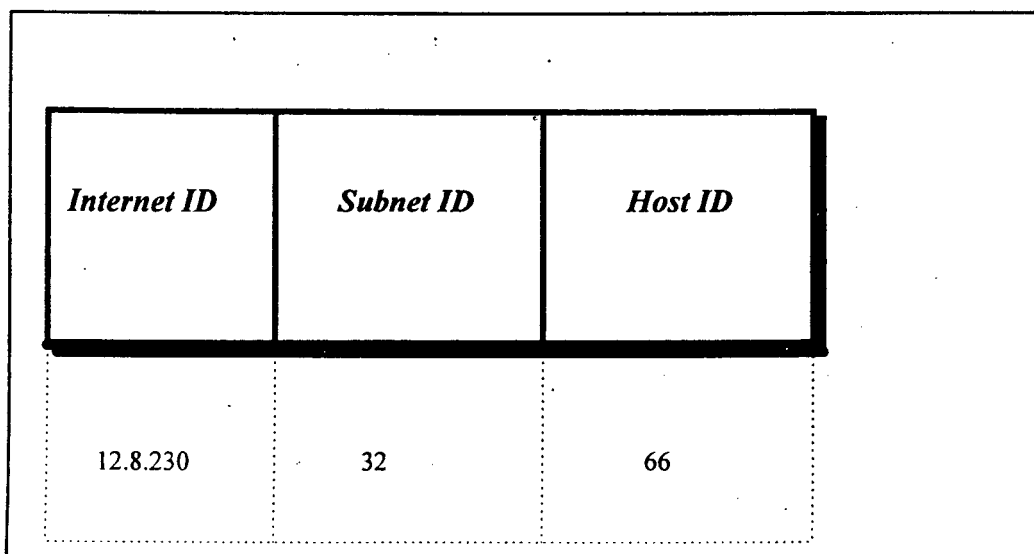


Fig. 3.3 *Conceptual representation of Internet addressing*

### 3.4 Sockets

To provide an interface between our application and the network we need a socket. Most of the communication in a client-server application is point-to-point, where the endpoint of such communication is an application (client or server). A socket acts as an endpoint for communication between processes on a single system or on different system. The applications communication between processes on a signal system or on different system. The applications communicate between themselves by sending messages to one another. These messages are sent as sequence of packets at the network level. For each packet that is sent, there has to be a receiving end. Sockets from such an end point to receive packets, as well as to send messages. Application programs request the operating system to create a socket when in need. The system returns a socket identifier, in the form of a small integer that the application program uses to reference the newly created socket. A networking application can be identified by a <host, socket> pair (the host on which it is running and the socket at which it is listening for messages).

Java provides separate classes which encapsulate the functionality of client and sever sockets. The Socket class is used to represent sockets on the client side, while the server side sockets are represented by the ServerSocket class. The Socket and ServerSocket form the client and server side sockets in a connection oriented protocol. Once a link is established between the client and the server, can exchange messages until one of them closes the connection. Whereas, the sockets, in the case of connectionless protocol, are represented by the DatagramSocket class.. In this case, both the client and the server are associated to a datagram socket. Every time a message is to be sent, they

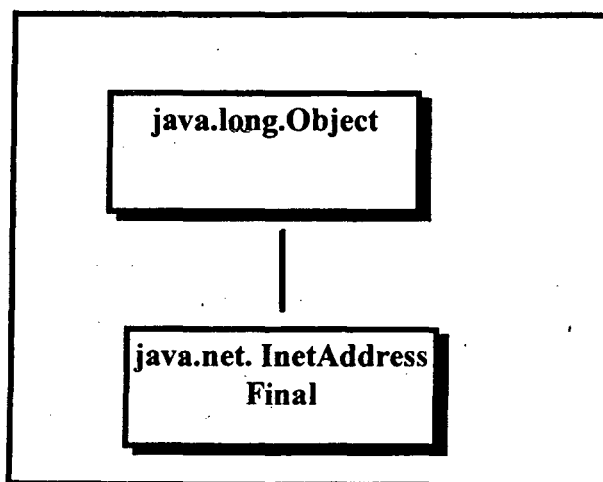
create a DatagramPacket containing the destination address and port number along with message to be sent. This packet gets delivered (if to does get delivered) to the targeted application. To implement various such policies for communication between a client and a server, Java provides SocketImpl Class SocketImplFactory is an Interface that can be used to generate more instances of the SocketImpl Class for use in our applications.

### 3.5 Classes And Interface Necessary For Developing Network Application using java

#### Class and interface description

<b>Class/Interface</b>	<b>Description</b>
<b>Inet Address</b>	Represents Internet addresses.
<b>Server Socket</b>	This class represents the server socket. It uses a Socketimpl class to implement the actual policies regarding socket operations.
<b>Socket</b>	this class represents the client socket. It uses a socketimpl class to implement the actual socket policies for its operations.
<b>Datagram Socket</b>	This class represents a datagram socket, which is an implementation of the connectionless protocol.

- Datagram Packet** This class represents a datagram packet, which is self-contained, with the details of the destination host and the data to be sent.
- SocketImpl** This should be subclassed to provide actual implementation. This class implements the actual socket policies for ServerSocket classes.
- SocketImpl Factory** A Factory for creating actual instances of socketImpl is defined in this interface.
- Purpose** Use InetAddress to represent Internet addresses.
- Syntax** Public final class InetAddress extends Object
- Description** The InetAddress class represents the Internet Address. The methods of InetAddress provide functionality to gain information about raw IP address, hostname, and network address of a host machine, and hash code of the Internet address in the hashtable.



**Fig. 3.4 Class diagram for InetAddress class**



## **EQUALS (OBJECT)**

**Class Name** InetAddress

**Purpose** Compares the specified object with the object on which the method is invoked.

**Syntax** public Boolean equals (Object object)

**Parameters**  
object

**Description** The object with which the invoked object is to be compared. the method compares the Internet address of the specified object with that of the object on which the method is invoked. The objects are considered equal if their Internet addresses are the same.

## **GETADDRESS()**

**Class Name** InetAddress

**Purpose** This method returns the raw IP address of the object in network byte order.

**Syntax** public byte[] getAddress()

**Parameters** None.

**Description** This method returns the raw IP address representation of the Internet address in 32-bit format in network byte order. It returns the addr[] byte array, member of the InetAddress. addr[0] contains the highest order byte position addr[] is an array of bytes so that this method is extendable for 64-bit IP addresses also.

## GETALLBYNAME(String)

**Class Name** InetAddress

**Purpose** Returns an array of all InetAddresses that correspond to the specified hostname.

**Syntax** `public static synchronized InetAddress[] getAllByName(String host_name) throws UnknownHostException`

**Parameters** **host\_name** The hostname of the machine, the InetAddresses which you are trying to obtain.

**Description** A host can have multiple InetAddresses (mapping). to access all of those InetAddresses, the hostname is passed to the getAllByName method. The method finds out the Internet addresses of the given host and returns all of them as an array of InetAddress objects.

## GETBYNAME (String)

**Class Name** InetAddress

**Purpose** This method gets the InetAddress of the specified host.

**Syntax** `public static synchronized InetAddress getByName (String host_name) throws UnknownHostException`

**Parameters** **host\_name** The hostname of the machine whose InetAddress is returned by this method.

**Description** This method returns the InetAddress of a specified host. The InetAddress class does not have public on structure. You can use this method to create an instance of the InetAddress for a particular host.

## **GETBYNAME(String)**

**Class Name** InetAddress

**Purpose** This method gets the InetAddress of the specified host.

**Syntax** public static synchronized InetAddress getByName (String host\_name)  
throws UnknownHostException

**Parameters**

**host\_name** The hostname of the machine whose IneAddress is returned by this method.

**Description** This method returns the InetAddress of a specified host. The InetAddress class does not have public constructors. You can use this method to create an instance of the InetAddress for a particular host.

## **GETHOSTNAME()**

**Class Name** InetAddress

**Purpose** this method returns the hostname for this InetAddress.

**Syntax** public String getHostName()

**Parameters** None

**Description** The method returns the hostname of a machine with the same address as this InetAddress. so if you know the IP address of a machine, you can find out its hostname by using this method on they InetAddress object of that address.

## **GETLOCALHOST()**

**Class Name** InetAddress

**Purpose** This method gets the InetAddress of the local host.

**Syntax** public static InetAddress getLocalHost() throws UnknownHostException

**Parameters** None

**Description** This method finds the Internet address of the local machine executing this program. It creates an instance of InetAddress with this address and returns the InetAddress object. getLocalHost() can be used to create an instance of InetAddress for the local machine.

## **HASHCODE()**

**Class Name** InetAddress

**Purpose** Returns the hash code of this InetAddress object.

**Syntax** public int hashCode()

**Parameters** None

**Description** The method returns the hash code, to be used as index into the hashtable to access this InetAddress object. All the InetAddresses accessed during the program execution are cached in a hashtable. This is done for faster access of previously accessed Internet address.

## **TOSTING()**

**ClassName** InetAddress

**Purpose** This method converts the InetAddress to a string

**Syntax** public String toString()

**Parameters** None

**Description** This method converts the InetAddress to a String by overriding the toString() method of the Object Class. Raw IP address, host name can also be obtained by manipulating with the returned String.

## SERVERSOCKET

**Purpose** Use ServerSocket to implement a server.

**Syntax** public final class ServerSocket extends Object

**Description** The ServerSocket class represents the server in a client-server application. This class implements the actual socket policies that go along with a server. It uses a default SocketImpl class to implement its server policies. These policies can be changed by implementing a concrete subclass of the abstract SocketImpl class. This change in policies can be made effective by setting the SocketImplFactory, using the setSocketFactory method. The methods of ServerSocket class provide functionality to create server socket, accept connection from a client and get the specific of the particular ServerSocket object (namely the port to which the server is connected), and the string form of implementation address, file descriptor, and port. A ServerSocket object is bound to the local machine on which it is created. A port number is specified for the ServerSocket to bind and listen for connection.

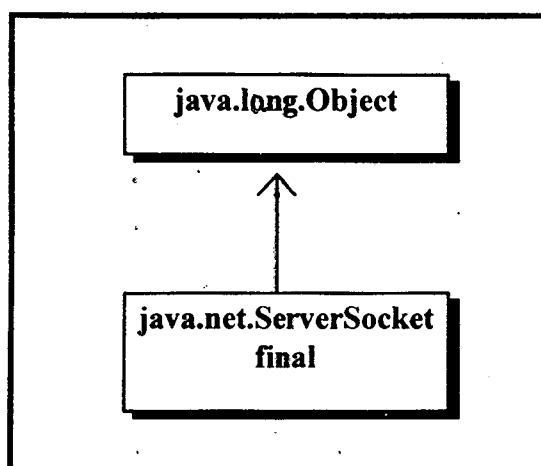


Fig. 3.5 Server Socket Class

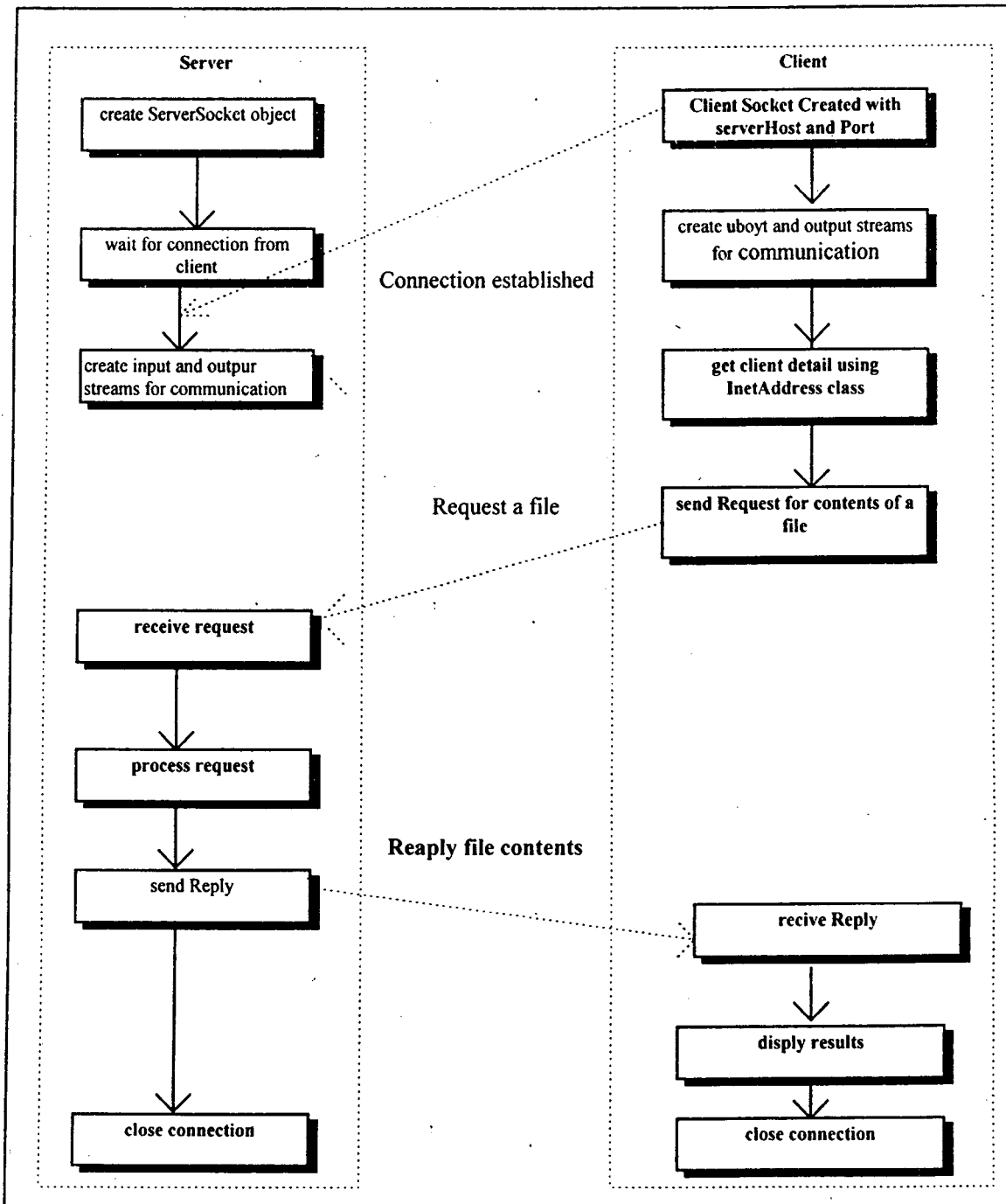
# CHAPTER - 4

## 4.1 Detail Analysis Design & Implementation for A Client-Server Information Applet Information

In the Client-Server Information applet, a client will contact an existing server and obtain details. We should provide three buttons : clientinfo, serverinfo, and fileinfo. On clicking the clientinfo button, the client will print out its details, the host it is running on, and the port on which it is connected. It should be able to gather the information about the server and print the details when the serverinfo button is pressed. When the fileinfo button is pressed, the client will send a request to the server to obtain the contents of a file residing on the server's side.

The applet should also be a stand-alone application, so that you can run it from a Java environment using the Java interpreter or it can also be launched from a Web browser, supporting Java. The scenario for this applet is illustrated in Figure 4.1. The steps involved are as follows .

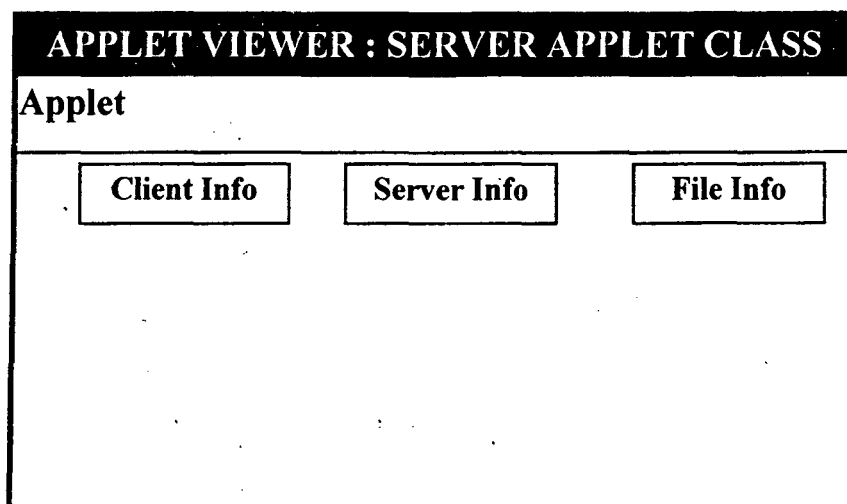
1. The server connects to a port on the host, to which it is initialized, and listens for connections from clients.
2. A client will connect to the server by specifying the hostname and the port on which the server is listening.
3. The client retrieves the details about itself and displays the information.
4. The client obtains the information about the server and displays it.
5. The client sends a request to the server asking for the contents of a file on the servers side.
6. The server, on receiving the request, opens the relevant file, if it exists, and sends the contents to the client over the socket streams.
7. The client receives the file contents from the server and displays them.



**Fig. 4.1** Sequence of actions in the Client-Server Information

8. Once the necessary processing is completed, the steams and sockets and are closed appropriately

We must now be able to implement the applet using the APPI. As we know, any applet for a given specification can be implemented in different ways. Once such implementation is provided here. In this implementation, Threads are used on the server's side to process multiple clients. In the given applet, the Server is started on a given machine. The Client can either be run as a stand-alone application using the Java interpreter or launched from a Web browser supporting Java. In either case, the server hostname and port number are passed as arguments to the executable.



#### 4.1.1 Building Our Applet

1. First create a Server class which should accept connection from client. Make it public class and the filename should be Server.java. This class should contain the following members



<b>Modifier</b>	<b>Type</b>	<b>Variable-name</b>	<b>Purpose</b>
Static	ServerSocket	ServSock	An instance of ServerSocket is created to enable the server to accept connections from client.
Static	Socket	theSocket	An instance of Socket that will be created on theServer side when Serversock is instantiated.
Static	int	Port	Port number on which the sever connects and creates SverSock
static	ServerThread	client[]	An array of ServerThread objects which will act as server for every client that connects to the server, so that there is one SeverThread for one client.

After the addition of these member our file Server.java should contain the following.

```
//import the necessary classes relevant to the class Server
```

```
import java.net.Socket;
```

```
import java.net.ServerSocket;
```

```
import java.io.*;
```

```
/**
```

```
public class Server {
```

```
    private static Socket theSocket;
```

```
    private static ServerSocket ServSock;
```

```
    private static ServerThread client[] = new ServerThread[10];
```

```
    private static int port = 80;
```

2. Having created the class, now define the main method for the class. Create an instance of ServerSocket after specifying the port number, Then the sever waits in a loop for connections from clients. This is done using the accept() method of ServerSocket. After accepting connection from a client. a server thread is spawned for each client. This enables exclusive service to the client by a corresponding

server Thread. The start() method of Thread will start the servicing of the server Thread to the client. Adding these functions to our Server class will make the Server.java file.

```
import java.net.Socket;

import java.net.ServerSocket;

import java.io.*;

public class Server {

    static Socket theSocket;

    static ServerSocket ServSock;

    static ServerThread client[] = new ServerThread[10];

    DataInputStream datain;

    DataOutputStream dataout;

    static int port = 80;

    public static void main(String srgs[]){

        int g=0

        try {

            SerSock = new ServerSocket(port);

            System.out.println("Server started");
```

```

While (true){

    theSocket = ServSock.accept();

    for (g=0; g<10;g++)

        if ((client[g] == null) || (!client[g].isALive()))

            break;

    if (g<10){

        client[g] = new SrverThread(the.Socket,g);

        client[g].start();

    }

    else

        System.out.println("Rejected a connection");

} catch (IOException ioe){

    System.out.print("Server error");

}

}

}

```

3. Now we have the necessary code for a Server to run. Next we should write the necessary code for implementing the ServerThread the Server spawns for every

Client. Create a public class ServerThread. It should be a subclass for Thread class in Java and should contain the following members.

Type	Variable name	Purpose
Socket	mySocket	This Socket object will be the socket instance on Server side for a connected Client. This is passed on to this ServerThread using the constructor.
int	myId	Every ServerThread has an id associated with it, which also identifies the Client. There can be only one ServerThread with a given Id, at a given instance.
DataInputStream	datain	DataInputStream associated with every ServerThread (which will be the Client's OutputStream) to receive messages for the Client.
DataOutputStream	dataout	DataOutputStream associated with every ServerThread (which will be the Client's inputStream) to send messages to the Client

With these data members and the constructor of the ServerThread class, our ServerThread.java should now contain the following.

```

import java.net.Socket;

import java.io.*;

import java.awt.*;

public class Server Thread extends Thread {

    private Socket mySocket;

    private DataInputStream datain;

    private DataOutputStream dataout;

    private int myId;

    public ServerThread(Socket m, int Id)    throws IOException {

        mySocket = m;

        myId = Id;

        datain = new DataInputStream(new
BufferedInputStream(mySocket.getInputStream()));

        dataout = new DataOutputStream(new
BufferedOutputStream(mySocket.getOutputStream()));

    }

}

```

4. Now we should override the run() method of Thread class in ServerThread class  
Assuming you have a function processRequests() in this class, the run method

will call the process Requests() while there are more requests from theClient. In the meantime, after processing every requires, the Thread should yield to the other ServerThreads to process their respective Client request. After all the request are processed we should close the sockets and streams that are open. Enter the following method into the ServerThread class.

```
public void run(){
try {
while (processRequests())
    yield(); // yield to other threads too!

    cleanUp();
} catch (IOException e){
    System.out.println ("Error in processing request");
}
}
```

```
void CleanUp() {
try {
    detain.close();
    dataour.clsoe();
}
```

```

mySocket.close();
catch (IOException io){
    printOut(io.getMessage());
}
}

```

5. Let retrieving the contents of a requested file be a service provided by the ServerThread. When the ServerThread receives a message "File" from a Client through the DataInputStream, it understands that the Client is requesting a file to be retrieved and it expects another message from the Client indicating the name of the file to be retrieved. The ServerThread then reads the file and sends its contents using the DataOutputStream. If the message is "By", the ServerThread understands that the Client intends to close the session and so the method returns false. This makes the Server Thread's run() method terminate and so the ServerThreads get disposed. To achieve the described effect, include the following processRequests() method, whose return type is boolean. This method assumes the existence of a GetFile() method in this class.

```

private Boolean processRequests() throws IOException {
    try

```



```

        String req = datain.readUTF();
    if (req.equals("File")){
        String file = datain.readUTF();
        GetFile (file);
        return true;
    }
    else if (req.equals("Bye"))
        return false;
    else {
        System.out.println("Unknown service requested");
        return false;
    }
} catch(IOException ioe) {
    System.out.println(Error in input from Client");
    return false;
}
}

```

6. As a final part of our ServerThread class, we should now implement the GetFile() method. Given a filename, this method will first check to see if the file exists and if it does, whether it is readable. Then using the file, the method creates a DataInputStream by passing the FileInputStream as a parameter. Next the method

read the file line-by-line and sends the line contents to the Client using the DataOutputStream object named dataout. It follows the file contents with an End Of file message to the Client. Add the following code into theServerThread class.

```
private void GetFile(String file_name) {  
  
    // buffer to get all the Lines in the file  
  
    StringBuffer buff = new StringBuffer();  
  
    file f = new File (file_name);  
  
    boolean b = (f.exists() || f.canRead());  
  
    if (!b)  
  
        printout("File either doesn't exists or is unreadable");  
  
    try {  
  
        DataInputStreamf_in = new SataInputSteam(new  
  
            BufferedInputSteam(new  
  
                FileInputStream (file_name)));  
  
        While (f_in.available() !=0) {  
  
            String line = f_in.readLine();  
  
            buff.append(line + "\n");  
  
            dataout.flush();  
  
        }  
  
    }  
  
}
```

```

dataout.writeUTF(EndOfFile");
dataout.flush();
} catch (IOException ioe) {
System.out.print(Error in handling file");
}
}

```

7. The Client class is the next one to be created Client is the class that will be launched as an applet from the Web browser. So it extends the applet and in this implementation, implements the Runnable interface. It acts as a client requesting service from an existing Server. According to the specification, this should also as a stand-alone application. It has instances of Socket, DataInputStream, DataOutputStream, and Thread as its members. It should also implement user interface with three buttons : clientinfor, serverfo and fileninfor.

8. Time implement this as a stand-alone application you need to write a method main(). the following code listing implements this method, which obtains the server name and port number from the command line. An instance of Client is created and a method myinit() is invoked to pass the parameters to the Client object. Then the Client Thread is started and a Frame is initialized to contain the

three buttons to be created Enter the following code in the client class to extend it as a stand-alone application..

```
Public static void main(String args[ ] ) throws IOException {  
Frame f = new frame (Client-Server Information”);  
    // obtain the port number from second parameter on command line.  
  
int port = (new Integer (args[1])). intValue();  
    // convert it to an inter from its string value  
Client Clnt = new Client(); // initialize and start it  
client.start();  
  
f.add(“Center”, Clnt); // create the frame  
f.resize (600, 800);  
f.show();  
}
```

9. Now that we have Buttons in the Panel, we have to override the action () method so that appropriate action is taking when a button is pressed. The following code achieves this. Enter the code in the class Client. Also include two variable, count will keep track of the lines printed out to the canvass.

```

String InpStr[ ];

int count;

public Boolean action (Event evt, object arg) {

if(evt. target instanceof Button) { // if a Button is pressed

count = 0;

if ("clientinfor".equals(arg)) { // if client button is selected

int port_num = sock.getPort();

printOut("Client has connected to a Server listening at the port number" + port_num);

ClienetInfor();

printOut("\n\n");

}

else if("serverinfor".equals(arg)) { //if serverinfo is requested

    ServerInfo();

    printOut("/n/n");

}

else if("fileinfor". equals(arg)) { // if fileinfor is requested

try {

    // Wr requesting the contents of file by name /etc/motd String file_name = new

String("/eetc/motd");

MakeRequests(file_name);

```

```

} catch (ArrayIndexOutOfBoundsException a) {

printOut ("For accessing remove file \n\tUsage : java client <filename>");

}

printOut(\n \n"); // a pretty print method available within this class

}

}

return true;

}

```

10. Information about the Client is to be retrieved when the clientinfo button id pressed. Including the following method in Client class will make this happen. If the Inet address of the local host is made available, then more details can be obtained from the InetAddress instance that will reflect the client machine information.

```

/** Method to obtain information about the client host */

public void clientInfo() {

    InetAddress c_inet;

    String c_name;

    try {

        c_inet = InetAddress.getLocal (Host ( ));    // InetAddress of the Local host

        c_name=inet.getHostname ( );    // Get the host name of the client

        printOut("Clent Host Details ");
    }
}

```

```

printOut(" HostName : " + c_name);

// get the string form of the inet address and extract the IP address part of it

String c_str = c_inet.toString ();

int index = c_str.indexOf ( '/' );

String c_ipaddr = c_str.substring (index+1);

printOut (" IP address : " + c_ipaddr );

} catch (IOException IOE);

}

```

11. Server details can be obtained from the server's `InetAddress` in a similar manner as from the Client's. To get the `InetAddress` of the server, the `Socket` instance is used. The `getInetAddress ()` method of `Socket` class is used. Include the following code in the Client class.

```

/** Method to obtain information about the server */

public void ServerInfo() {

    InetAddress s_inet = sock.getInetAddress();

    String s_name = inet.getHostName ();

    printOut (" Server Host Details ");

    printOut ("HostName : " + s_name);
}

```

```
String s_str = S_inet.toString();

int index = s_str.indexOf ( ' / ' );

String S_ipaddr = s_str.substring (index+1);

printOut(" IP Address : " + s_ipaddr );
```

12. The following Make requests () method is used to send the file name to the server and request the contents of the file. The client then reads the reply from the server and prints it on the screen until the end of file is reached. The datain and dataout members of type DataInput Stream and DataOutputStream are used by the Client to communicate with the Server.

```
public void MakeRequests(String fil_n) {

printOut( "File requested by the client : " +file_n);

printout (" \n \n");

printOut( The file contains the following :");

printOut("\n \n);

try {

    dataout.writeUTF(File");    // informa the server that youare requesting a file

    dataout.flush();           // send the filename to the server

    dataout.writeUTF(fil_n);    // always a use flush() after using write() method of

                                // outputstream
```



```

String file_contents = datain.readUTF();

while (!file_contents.equals("EndOfFile")){

printOut(file_contents);

file_contents = datain.readUTF();

dataout.writeUTF("Bye");

data out.flush();                // transaction is complete

dataout. flush();

}    catch (IOException ioE){

// catch the i/o exception

System.out.println("Oops! file prob");};

```

13. Include the following methods in the class client, To print the strings on the canvas in an orderly manner, write the printOut() method, which is used by other methods to print on the canvas. The paint() method is overridden here to write to the exact locations the canvas.

```

public void printOut(String str) {

    InpStr[count] = new String(str);

    count++;

    repaint();

    try{

        mythread.sleep(500);

```

```

        }catch (InterruptedException ie) {};
    }

    public void paint(Graphics g) {
        dimension d = size();
        g.setColor(color.black); // write the contents in black

        for (int y=60, i = 0; i <count; i++ {
            y +=20; // between each line leave 20 pixels gap

            // draw the string at 40th columns and specified line 'y' g.drawString(InpStr[i],40,y);
        }
    }
}

```

14. We should take necessary care to close any open files, steams, or sockets. This can be done in the stop() method of the Applet, which is called when is called when the Applet is terminated.

```

public void stop() {
    System.out.println("inside Client.stop()");

    if (mythread != null) {
        mythread.stop();
        mythread.stop();
        mythread=null;
    }
}

```

```

        try {
            dataout.close();
            datain.close();
            sock.close();
        } catch (IOException E);
    }
}

```

15 The above three files are compiled using javac. The server is executed using the Java interpreter. Use

**java Server**

At the command prompt to run the Server.

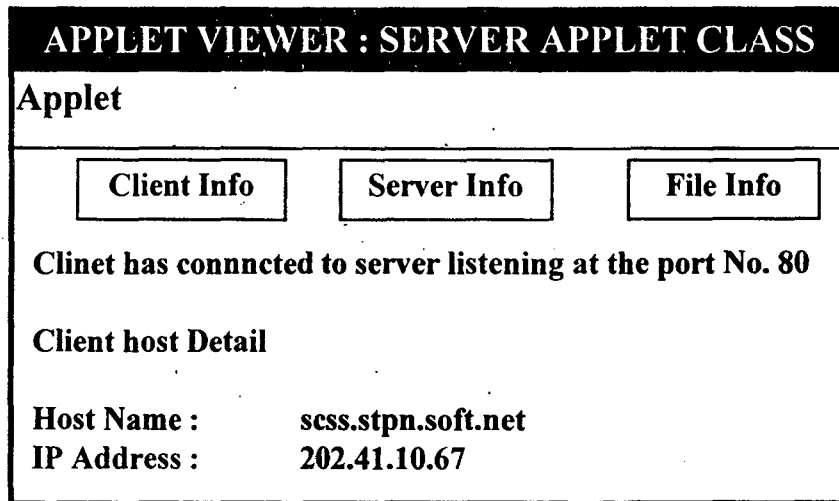
16. The client Applet can be launched from the Web using the following HTML file, csr.html.

```

<title> Client-Server Information </title>
<hr>
<applet code = Client.class width = 600 height = 400>
<param name = servPort value = 80>
</Applet>
<hr>

```

The applet, when launched, using the command `applet viewer csr.html`, will create Panel what will appear as in Figure . When you press any of the three keys, appropriate actions taken and details are printed out canvas. This applet implements the Client-Server information exchange of information between the Client and the Server.



#### 4.1.2 How the The Applet Works

The server is a stand-alone application. First, start the server on the host you want to run the server. After starting the server, run the client applet. When the applet comes up, it displays a window with three buttons : `clientinfor`, `serverinfo`, `fileinfor`. If you click the `clientinfor` button, the details of the host, on which the client applet is executed, is displayed on the canvas. If you click the `serverinfo` button, the details of the server host, host on which the server is running, is displayed on the window. Where as if you click on the `fileinfor` button, the contents of the `etc/motd` file (in case of Unix system) is displayed on the screen. If you are interested in any other file, changes the filename in the code to the desired filename.

## **4.2. DETAILS ANALYSIS DESIGN AND IMPLEMENTATION IN JAVA FOR CHAT SERVER PROTOCOL**

### **4.2.1 UNDERSTANDING CHAT AREAS**

Ever since the first computers were connected, people have been using them to talk. A *chat area* is an interface that lets a group of people talk by typing messages. Like all Internet based programs, chat programs must follow a specific protocol. Specifically, chat programs rely on the internet chat (IRC) protocol. However, chatting on a Web page has been ad-hoc at best until Java. With Java, a web page can have fully featured chat areas. Features such as instant messages and membership rooms are not difficult to implement. Due to the applet restrictions, chat applets on the same page will have to communicate through a server on the computer where they reside.

### **4.2.2 CREATING OWN CHAT PROTOCOL**

To create own chat area using Java, we must have to define protocol. Protocol allows a client and server to communicate across a network. To write our own protocol we must have to define the information that the client and server will exchange. For a chat area, the client must be able to send its name, receive the names of the people in the chat area, send messages that appear on the chat area's message board. The server must be able to update the client list when someone enters or leaves the room, or

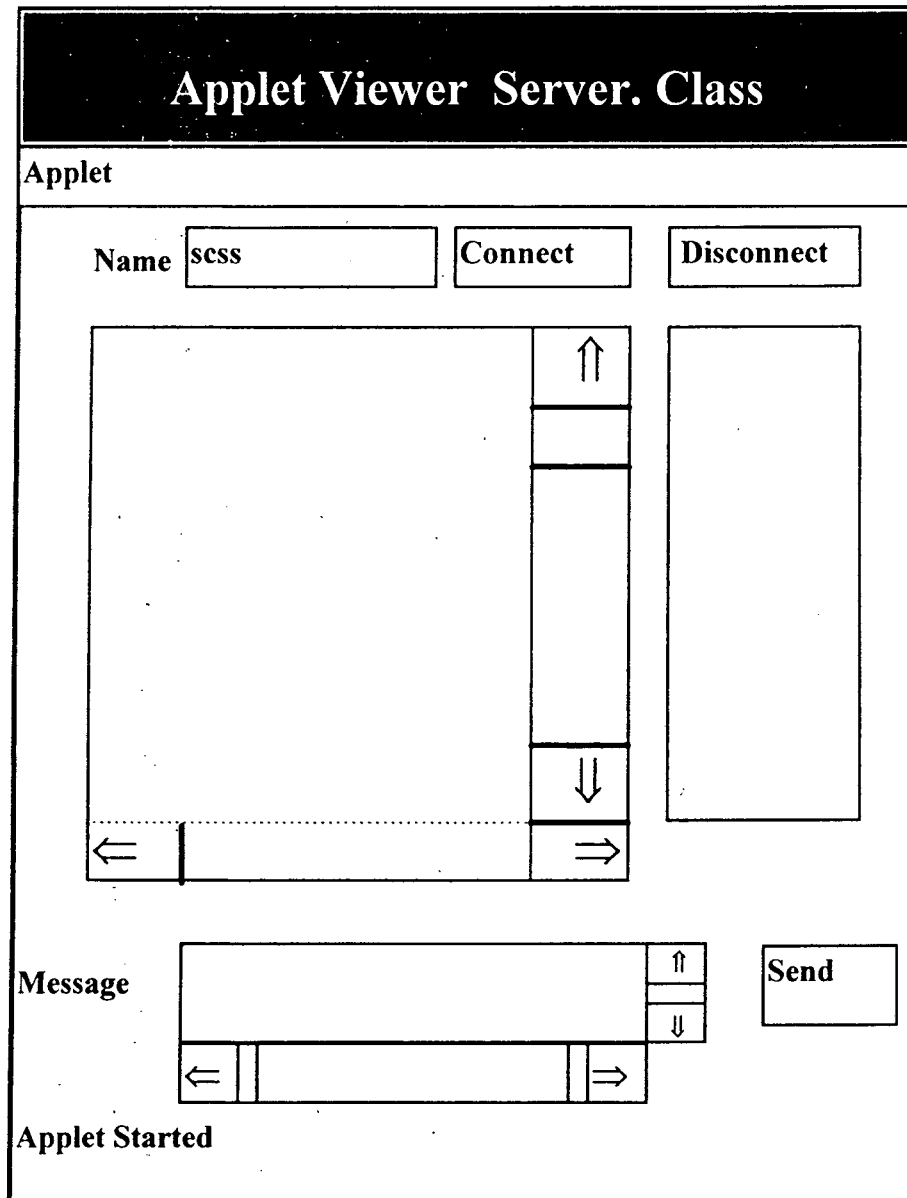
when there is a new message. Following table shows a simple protocol for a chat area client-server application.

<u>Protocol</u>	<u>Client</u>	<u>Server</u>	<u>Description</u>
Hi name	yes	no	connect to a chat server.
TAKEN	no	yes	connection refused because name is taken.
PEOPLE:name:name:.....	no	yes	list of every one in the chat area.
MSG: message	yes	yes	Message to be placed on the chat area board.
QUIT	Yes	yes	close the connection nicely.

### 4.2.3 BUILDING A CHAT APPLET'S INTERFACE

A minimum chat applet has two text areas, a text field, a list, and three buttons. The text field and one button are used to create a name and connect to the chat server. One of the text areas and another button are used to send messages to the chat server the other text area is used to display messages that are received from the chat server. The last button is used to disconnect from the server. The interface also has a list to show all the people chat that are connect to the server.

To implement in java for the chat interface we have to use event handling concept.



First of all define a chat applet class like

```
public class chatapplet extends Applet{
public void init()
```

```
{  
  
}  
  
}
```

#### 4.2.4 HANDLING THE CHAT APPLET'S EVENTS

The chat applet is an event driven program. we must implement the applet's action method to capture the button - down events . When the user clicks on the connect button , the chat applet tries to connect to the chat server After the applet establishes a socket connection, the chat applet sends its name to the server. when the user clicks on the disconnect button , the chat applet closes the connection.

In addition , if the chat applet is connected to a server , the user's selection of the send button causes the applet to send the contents of the message-text area to the chat server. The following code demonstrates how to process a *chat applet events*.

```
public boolean action(Event evt,Object obj)  
{  
    if(evt.target instanceof Button)  
    {  
        String label= (String) obj;  
        {  
            if (label.equals(connect))  
            {  
                if (soc==null)
```



```

{
try
{
soc= new Socket(InterAddress.getLocalHost(), no );
ps= new printStream(soc.getOutputStream());
ps.println(name-text.getText());
ps.flush();
listen=new Listen(this ,name-txt.getText(),soc);
listen.start();
}
catch(IOException e)
{
System.out.println("Error:"+e);
disconnect();
}
}
}
else if (label.equals(DISCONNECT))
{
disconnect();
}
else if (label.equals(SEND))

```

```

    {if (socket!=null)
    {
    StringBuffer msg=new StringBuffer("MSg: ");
    ps.println(msg.append(msg-txt.getText()));
    ps.flush();
    }
    }
    }

    return true;

    public void stop()
    {
    disconnect();
    }

    public void disconnect()
    {
    if (soc!=null)
    {
    try
    {
    listen.suspend();
    ps.println("QUIT");
    ps.flush();

```

```

soc.close();

}

catch(IOException e)
{
system.out.println(Error:" +e);
}

finally
{
listen.stop();

listen=null;

soc=null;

list.clear();
}
}
}

```

In the above code Listen class within the code.

The listen class is a thread that listens to the *chat server*.

#### **4.2.5 PROCESSING MESSAGE RECORD FROM A CHAT SERVER**

An event - driven program can also receive events from other programs. For chat applet program, we must handle messages received from the chat server . There are three messages that the chat applet might receive from the chat server.

When someone enters or leaves the chat area , the chat server sends the PEOPLE keyword followed by a list of names .When client sends a message to the chat area , the chat server sends a MSG keyword followed by the message. Finally, when the chat server is going to disconnect , it sends the QUIT keyword. The following classes demonstrates how to handle messages received from a chat server.

```
class listen extends Thread {  
  
    public Listen(chatapplet p,String n, socket s)  
  
    {  
  
    -  
  
    --  
  
    --  
  
    --  
  
    try{  
  
    --  
  
    }  
  
    catch (IOException e)  
  
    {  
  
    }  
  
    }  
  
    public void run()  
  
    -  
  
}
```

```
while (true)
{
    try{
        msg= dis.readLine();
    }
    catch (IOException e)
    {
        -
    }
    if(msg=null)
    {
        -
    }
    if(keyword.equals("PEOPLE");
    {
        chatApplet.list.clear();
        while(st.hasMoreTokens())
        {
            -
        }

        else if (keyword.equals("MSG"))
```

```

{
--
}

else if(keyword.equals("QUIT"))

{-
-}

}

}

}

```

The Listen class extends to simplify the design of the chat applet also improves the response to user input Using threads , the chat applet can detect new messages being sent from the server and take input from the user at the same time .

#### 4.2.6 HOW A CHAT SERVER ACCEPTS A CLIENT

The first task of a chat server is to creat a server socket on a specific port . Then the chat server must wait for clients to connect. When a client request a connection , the chat server creats a thread for that client , saves the client with a client list , and then waits for the next clients The following code demonstrates a chat server main method , where a chat server waits for a new client.

```
public class chatserver extends Frame{
```

```

static Vector clients = new Vector(10);

static server Socket server =null;

static int active - connects =0;

static Socket socket =null;

public boolean handle eventEvent(Event evt)
{
    if (evt.id ==Event.WINDOW-DESTROY)
    {
        sendClients(new stringBuffer("Quit");
            closeAll();
        System.exit(0);
    }
    return super .handleEvent(evt);
}

public static void main (string args[ ])
{
    Frame .f= New chatServer();
    f.resize(200,200);
    f.show();
    try{
        server=new serversocket(2523);

```

```

    }

    catch(IOException e)
    {
        System.out.println("Error" " e);
    }

    while (true)
    {
        if(clients.size()<10)
        {
            try{
                socket=server.saccept();
            }

            catch(IOException e)
            {
                System.out.println("Error:" +e);
            }

            for (int i=0; i<chatserver.client.size();i++
            {
                client c= new client(socket);

                clients.elementAt(i)=c;

                if checkName(c))
                {

```



```
c.start;
notifyroom();
}
else {
c.ps.println("TAKEN");
dosconnect(c);
break;
}
}
else
{
try{
Thread.sleep(200);
}
catch(InterruptedException e){}
}
}
}
}
```

#### 4.2.7 CREATING A CHAT SERVER 'S CLIENT THREAD

How to create chat server that lets multiple clients make a connection. For each of these clients, the chat server must be able to accept messages. The best way to do this in java is to create a separate thread per client. The following class demonstrates how to connect multiple clients.

```
class Clients extends Thread{  
  
    public void send (String buffer msg)  
  
    {  
  
    ps.println (msg);  
  
    ps.flush();  
  
    }  
  
    public clients (socket s,int i)  
  
    {  
  
    socket =s  
  
    try{  
  
    dis= new DataInput Stream(s.getInputSream());  
  
    ps= new printstream(s.getOutputStream());  
  
    ps= new printstream(s.getoutputStream());  
  
    name= dis.readLine( );  
  
    }  
  
    catch (IOException e)
```

```

    {
        System.out.println("error:" e);
    }
}

public void run()
{
    while(true)
    {
        String line =null;
        try
        {
            line= dis .readLine( );
        }
        catch ( IOException e)
        {
            System.out.println("error: " +e)

            chatserver.dissconnect(this);
            chatserver.notify.Room();
            return;
        }
        if(line=null)
        {

```

```

chatserver.disconnect(this);

chat.server.notifyRoom();

return;
}

stringTokenizer st=new StringTokenizer(line, " ");

String keyword= st.nextToken();

if (keyword.equals("MSG"))

{

stringBuffer msg= new string Buffer ("msg:");

msg.append(name);

msg.append(st.nexttoken("\0");

chatserver.sendclients(msg);

}

else if (keyword.equals("QUIT");

{

chatserver.disconnect(this);

chatserver.notifyRoom();

this.stop();

}

}

```

```
}
```

```
}
```

#### 4.2.8 IMPELEMENTING CHAT SERVER METHODS

Some of the requests require the server to send the messages to all other clients that are connected to the chat server. For example if a client sends a QUIT message, chat server must notify all other clients that one of the client has left. Likewise, if a client sends a MSG, the chat server must send all other clients that message.

When a chat server gets a connection request from a new clients, the chat server must make sure that the name is not taken away by another clients.

The following code illustrates the processing the server must perform

```
public static void notifyRoom()  
  
String Buffer people = new String Buffer ("PEOPLE");  
  
for(int i=0; i<clients.size();i++);  
  
{  
  
clients c =clients .elementAt(i);  
  
people.append(":" c.name);  
  
}  
  
sendClients(people);  
  
}  
  
public static synchronised void send Clients (StringBuffer msg)  
  
{
```

```

for (int i=0; i<clients.size(); i++)
{
    clients c= clients .elementAt(i);
    c.send(msg);
}
}

public static void closeall()
{
    while (clients.size())
    {
        client c= clients firstElement();
        try{
            c. socket.close();
        }
        catch(IOException e)
        {
            system.out.println("Error: "+e);
        }
        finally{
            clients . removalElement(c)
        }
    }
}
}

```

```

}

public static boolean checkname(Client newclients)
{
for ( int i=0 ; i<clients.size(); i++)

client c= client.elementAt(i);

if((c !=new clients)&&c.equals(newclient.name))

return(false);

}

return(true);

}

public static synchronised void disconnect(Client c)
{

try{

c.send(new stringBuffer("QUIT"));

c.socket.close();

}

catch(IOException e)

{

system.out.println("Error:" +e);

}

finally

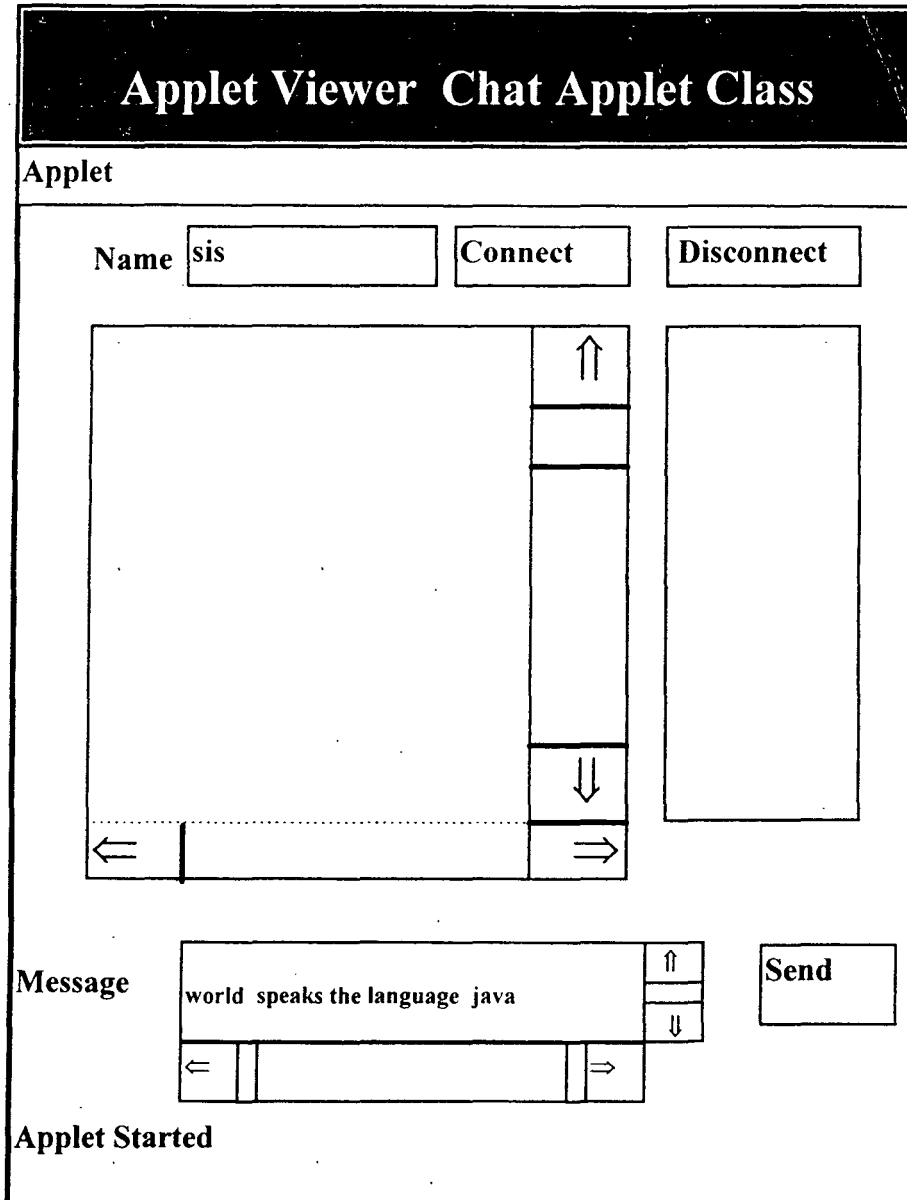
{

```

```
clients.removeElement(c);
```

```
}
```

```
}
```





# CHAPTER 5

## 5.1 CONCLUSION

A Chat Server has developed using JDK- 1.1. In the Chat area a group of people can talk by typing messages. The Developed Applet is interactive , dynamic and all also real time for any message. The Applet is to be accessed through Internet. Audio & Video can be included for real-time face to face talking.

Futures such as instant messages and membership rooms are also implemented. Due to the applet restrictions chat applets on the same page. will have to communicate through a server on the computer where they reside.

Running a Chat server on an existing intranet or extranet server allows companies to offer live interaction among employees and associates, and public chat areas let organizations host discussions across the Internet.

## 5.2 REFERENCES

- 1) [www.micr.co.uk](http://www.micr.co.uk)
- 2) [www.bcpl.lib.md.vt.edu/~frappa/pirch.html](http://www.bcpl.lib.md.vt.edu/~frappa/pirch.html).
- 3) A.S. Tanenbaum, *Computer Networks* ( Third Edn.), (PHI) 1997
- 4) Subhodh Bapat, *Object Oriented Networks, PTR Prentice Hall, 1994*
- 5) PC Magazine, *Can we chat?* May 27, 1997
- 6) IEEE Communications , Vol 35, no. 5 (May issue), no. 6(June issue), no. 10 (Oct. issue), 1997 (Complete issue)
- 7) Michale Morrison, *Unleashed Java* (Samsnet), 1995
- 8) P. Naughton , Herbert Schildt, *The complet references Java* (TMH), 1997