

132

Multilingual Indian Language Interface to Web

Dissertation Submitted to
Jawaharlal Nehru University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology
in
Computer Science & Technology

Submitted by
GAJANAN NIAL



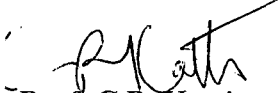
to the
School of Computer & Systems Sciences
Jawaharlal Nehru University
New Delhi-110067
INDIA
January, 2000

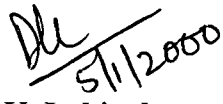
Certificate

This is to certify that the dissertation entitled **“Multilingual Indian Language Interface to Web”** being submitted by me to the **Jawaharlal Nehru University** in partial fulfillment of the requirements for the award of the degree of **Master of Technology in Computer Science and Technology** is a record of original work done by me under the supervision of **Dr. D. K. Lobiyal**, School of Computer and Systems Sciences, during the **Monsoon Semester, 1999**.

The results reported in this project have not been submitted in part or full to any other University or Institution for the award of any degree.


(GAJANAN NIAL)


Prof. C.P. Katti
Dean,
School of Computer &
Systems Sciences
JNU, New Delhi
INDIA


Dr. D.K. Lobiyal
Asstt. Professor
School of Computer &
Systems Sciences,
JNU, New Delhi
INDIA

Abstract

The rapid growth in the number of Internet users in India demands a large amount of content in Indian languages placed across the Internet. For communication across the globe Internet has served as one of the most efficient means. But, when it comes to communicating in local dialects and scripts, we find lot of limitations. Many users find it difficult to put forth their thoughts in a foreign language like English. Certainly, Internet users in a country like India would prefer to communicate each other in native languages and scripts. Moreover, to fill up forms for HTML format users have to depend on the Latin-1 encoding to submit the data. Keeping in mind the multilingual features of the Indian languages, we have provided a solution, which would allow users to fill up forms and communicate with each other through an online chat application in their native languages. Presently, our solution caters Hindi together with English as a means of communication, however it can be extended to any Brahmi based Indian script.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Dr. D. K. Lobiyal, Assistant Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University, for the unfailing support he has provided throughout. In all respects, I am grateful for the patience he has exhibited and for the time he has spent with me. It would have been impossible for me to come out successfully without his constant supervision.

I extend my thanks to Prof. C. P. Katti, Dean, School of Computer and Systems Sciences, J.N.U., for providing me the opportunity to undertake this project. I would also like to thank the authorities of our school for providing me the necessary facilities to complete my project in time.

I acknowledge and thank each and every one of those who directly or indirectly helped me in this work. Specially, a mention about my family members is a must, who have been source of support and encouragement for me. I would also take this privilege to thank all my friends, lab mates in JNU, and coworkers at UBF for their valuable support and concern for this work.

Last but not the least, I thank and praise God, who has been with me and guiding me perfectly according to his sovereign will and plan.

5th January, 2000
SC & SS, J.N.U.,
NIAL)
New Delhi.

(GAJANAN

Table of Contents

1 Introduction	1
1.1 Driving Force behind the Work	1
1.2 Problem Definition	2
1.3 Organization of the Report	3
2 Related Work	4
2.1 The need for Multilingual Indian Language Interface	4
2.2 History of Indian scripts	4
2.3 Structure of Indian scripts	5
2.4 Standardization a need	7
2.5 Text processing of Indian languages on typewriters and computers	8
2.6 Fonts for Indian languages/scripts	8
3 Character and Font Encoding for Indian Languages	11
3.1 Concept of Character Encoding	11
3.1.1 ASCII	11
3.1.2 ISCII	12
3.1.3 Unicode	13
3.1.4 UTF-8	13
3.2 Concept of Font Encoding	14
3.2.1 Problems faced in designing fonts for Indian languages	17
3.2.2 Fonts as used in web pages and web displays	18
4 Design and Implementation	20
4.1 A Solution through Java Applet	20
4.2 Configuration Files	20
4.2.1 Character Encoding format	21
4.2.2 Keyboard Map	21
4.2.3 Font Information	22
4.3 Devanagari Text Area	23
4.3.1 Documents	23
4.3.2 Keyboard to ISCII encoding	24
4.3.3 Keyboard Map	24
4.3.4 ISCII	24

4.4 Formation of Devanagari Characters	24
4.4.1 Generation of Characters	24
4.4.2 Generating Glyphs from Intermediate Display Sequence	25
4.5 Position of Matras	25
4.6 Caret Positioning	25
4.7 Text Manipulation	26
5 An Online Chat Application in Indian Language	27
5.1 Building of the Chat Applet	27
5.1.1 Submit a Chat	27
5.1.2 Reading the Chat data	28
5.2 Adding Indian language features to the Chat Applet	32
5.3 Sending Chat Data in Hindi	33
5.3.1 Typing Chat Data in Hindi	33
5.3.2 Encoding Chat data into UTF-8	33
5.3.3 SMTP	33
5.4 Receiving Chat Data typed in Hindi and English Dynamically	34
5.4.1 POP3	34
5.4.2 Decoding the Chat Data	35
5.4.4 Displaying the Chat Data	35
5.4.5 Online Chat in other Indian Languages	35
6 Conclusions and Future Work	37
6.1 Limitations	37
6.2 Future Work	37
Bibliography	39

Chapter 1

Introduction

The Internet has grown enormously in the recent past. Moreover, there is an exponential growth in the number of Internet users around the globe. In India, Internet users are increasing rapidly. But in a multilingual country like India where people find it difficult to communicate in a foreign language, it is necessary to enable the people to communicate in their own languages. This has resulted in large amount of content being created in Indian languages on the Web. People would prefer to use their native tongue to communicate, for example, sending e-mails in Hindi would be preferred than in English. Some software allows people to communicate in Indian languages, but they are mainly platform dependent and they require the user to install their package or a similar package. In this project work the aim is to provide an interface for Indian languages on the Web, which would help users to communicate in Indian languages without the explicit installation of any package. The solution provided is platform independent as it is written entirely in Java.

1.1 Driving force behind the work

As of now, none of the existing packages provide facility to fill forms in Indian languages, which would be quite handy for the users, who are not so proficient in English. Typically, one would like to provide a solution in which the software can be embedded in the User-Agent, so that the user need not bother about a viewer. Moreover, when it comes to filling of forms in a Web page, the solution provided by HTML is not sufficient for supporting Indian languages. The users have to type using the Latin-1 encoding scheme, which is not very convenient for typing in Indian languages. A typical scenario is that the designers of the Web page provide a transliteration scheme which would be used in the backend to process the user's request and give an appropriate response. Some of the disadvantages of this solution are:

- It is difficult for users to work with the transliteration scheme provided by the source.
- Different people can use different transliteration schemes.
- The user is never sure of whether the input he has given in the form is the same as he intended to.

Ideally, users should be able to type in their native languages, preferably with convenient key mappings and the characters typed should be displayed in the native language itself, so that the user need not worry about any ambiguity. In this project work, a Java applet has been provided, which can be embedded into an HTML document and thus can be used to fill up forms. Further, the information given as input through the applet allows easy processing, as the server-side software can be easily written. As an example application of the applet provided could be used to have a facility of on-line Chat and to send e-mails in Indian languages. Though we have provided a solution for Hindi along with English, it can be easily extended to any Brahmi based script. The solution requires the Java plugin to be installed on the client machine as our applet is written using JDK1.2 and also the font used by the applet, for rendering the glyphs, to be present on the client machine.

1.1 Problem Definition :

The enormous growth in the use of Internet in India demands various kinds of services provided by the World Wide Web. Specifically, communication across the globe through Internet has proved to be very successful and the most efficient way. However, when it comes to communicate in Indian languages, the present browsers do not provide enough scope for that. As a result, many users find it difficult to put forth their thoughts in a foreign language like English. Moreover, to fill up forms for HTML format, users have to depend on the Latin-1 encoding to submit the data.

Therefore, the need of a wrapper for Indian languages arises to allow users to fill up forms and communicate each other in Indian languages, supporting dynamic fonts. Such a wrapper will facilitate in editing and viewing text in Indian languages. The application

areas of having a facility like this will include sending and receiving mails, web publishing and online chat etc. in Indian languages.

1.2 Organization of the report :

The present chapter is the introductory part of the study. In chapter 2, we have discussed related work in the area of multilingual aspect of the Indian languages and their scripts. The standard of encoding characters and fonts for Indian language content is discussed in chapter 3. The design and implementation aspects for Indian language scripts are taken up in chapter 4. Chapter 5 presents the building of a Chat Applet in Java, which provides a facility for online chat in Hindi and English language script. In chapter 6, we have given the conclusion and future direction for the improvement of this work. The Appendix enumerates some sample configuration files.

Chapter 2

Related Work

2.1 The Need for Multilingual Indian Language Interface to Web

The demand for computer systems capable of providing input/output facilities in Indian scripts started in 1970s. The need for this facility was felt when the computers became popular for office automations. As a result of this lots of Research and Development efforts were made by various organizations to provide this facility. The approaches made by this people were mostly for the specific script and on ad-hoc basis. The simultaneous use of ASCII was not possible. The common formula for all the scripts in India was not thought. It was DOE (Department of Electronics) who took initiative in 1983 for standardization of the Indian code chart. This ISSCII-83 code complied with ISO 8-bit code recommendations (Report of the sub-committee on standardization of Indian scripts and their codes for Information processing, DOE July 1983). While retaining the ASCII character set in the lower half, it provided the Indian script character set in upper 96 characters. This also had the recommendation on a Phonographic based keyboard layout for all Indian scripts.

A common platform for all Indian scripts was thought in 1986 by DOE (Report of the committee for "Standardization of Keyboard Layout for Indian Scripts Based Computers" in Electronics-Information & Planning, Vol.14, No.1, Oct 1986). This report recommended 8-bit ISCII code. The revmsmon of the same in 1988 made the code chart more compact.

2.2 History of Indian Scripts

There are 15 officially recognized Indian scripts. These scripts are broadly divided into two categories namely, Brahmi and Perso-Arabic scripts. The Brahmi scripts consists of Devanagari, Punjabi, Gujarati, Oriya, Bengali, Assamese, Telugu, Kannada, Malayalam and Tamil. And the Perso-Arabic scripts include Urdu, Sindhi and Kashmiri. The Devanagari script is used for Hindi, Marathi and Sanskrit languages only. The characteristics of the languages within the family are quite peculiar. They have the

common phonetic structure, making the common character set. Within the same family again north Indian scripts like, Hindi, Marathi, Punjabi, Gujarati, Oriya, Bengali, Assamese have common features while Southern scripts like Tamil, Telugu, Kannada and Malayalam have common features. This clear division of characteristics has simplified the use of computers in a multilingual aspect.

All these scripts mentioned above are written in a nonlinear fashion. Unlike English, the size of the characters are different even within the same script. The division between consonant and vowel is applied for all Indian scripts. The vowels getting attached to the consonant are not in one (or horizontal) directions, they can be placed either on the top or the bottom of consonants. This makes the use of the scripts on computers more complicated to represent them.

To use language for any applications the characteristics of that language are required to be known. Once this is known, the application can make use languages in a most uniform manner. Indian scripts have a very different structure and have communality amongst them. They follow almost same rules, however, the way of representing them is different.

2.3 Structure of Indian Scripts

Since the origin of all these scripts is same, they share the common phonetic structure. The alphabet may vary slightly and also in their graphical shapes. All of them have basic consonants and vowels, their phonetic representation is also same. Using this characteristic a transliteration facility between any Indian scripts is possible. Same way it can be represented in Roman with the help of diacritic marks. Typically the alphabets are divided into following categories:

The Consonants

All Indian scripts use 5 types of consonants groups, called *varga*. Some of the vowel like "a" is included in the consonant category. Each *varga* has 5 consonants, with primary and secondary pairs. The second consonant in each pair is derived from the first consonant with 'h' sound, and has separate graphical representation.

Consonants				nasal	example
K	Kha	Ga	Gha	Na	gangA
Cha	Cha	Ja	Jha	N-a	manc
Ta	Tha	Da	Dha	N.	ghantA
Ta	Tha	Da	Dha	N	sant
Pa	Pha	Ba	Bha	M	stambha

other consonants not present in this category are,

Ya Ra La Va 'Sa S.a Sa Ha

and invisible consonant like, Ra (halant) and (halant) Ra, get formed differently.

Vowels

All the vowels are represented by separate symbols. These vowels are placed on the consonants either in the beginning or after the consonants. Each of these vowels are pronounced separately. Typical vowels are,

vowel : A i Ee u U ru Ee

usage : Ka Ki Kee Ku KU Kru Kee

vowel : e E a o O au ao

usage : Ke KE Ka Ko KO Kau Kao

anuswar

Gets used with nasal as shown in the example of consonants. e.g. rambhA

chandrabindu

e.g. PAnch

Visarga

Puts a sound of 'h' between two consonants. e.g.

du : kha

While forming the conjuncts, the use of broken consonants is activated by *halant*. On mixing of two or more consonants the shape of the conjunct varies. Many a times *halant* is required to indicate the vowel-less ending e.g. Ramnathan.

conjunct

Complex form made from consonants and vowels.

nishkriya = Na Ee Sha (halant) Ka (halant) Ra i Ya

Halant is also used to make the soft halant and explicit halant.

Ka (halant) Ta = Kta to make Ka (halant) (halant) Ta = Kat

Nukta

Nukta is used to derive some of the characters used in Hindi, Punjabi and Urdu etc.

Punctuation and numerals

All the punctuation and numerals are common between English and Indian scripts. They are used on the computers using English symbols.

Vedic characters

Apart from Hindi and Marathi, Devanagari has Sanskrit language, which uses Vedic symbols. The provision of these symbols is made by keeping the extended character set.

2.4 Standardization- a need

The applications of scripts on the computers have rapidly increased. These applications are of various nature which include, data processing, DeskTop Publishing, Telegraphic applications etc. If each of these application starts using the scripts in its own way then the chances of their interfacing between any two fields becomes rare. This interfacing requires enormous technical efforts and infrastructures to suit the requirements of both the applications. At the same time similar kind of applications might use the script in a different manner, which will again limit the use. All these efforts for each applications, and their limited use can be avoided by standardizing the script code. This code can be designed by taking care of the characteristics of the languages and their

uniform rules. The standardization of the code charts for Indian languages for computer applications has been done. This has made the implementations easier.

2.5 Text processing of Indian Languages on typewriters and computers

The text processing of Indian scripts on the mechanical typewriter and on computers is different. There are some limitations on the mechanical typewriter as compare to computers. Due to the complex nature of Indian script, formation of conjuncts is extremely difficult on the typewriters. The approach used in typewriters is most suited for graphical representation. For example, the formation of 'Pha' is done by using 'Pa' and remaining graphical part of 'Pha'. This is not a very user-friendly approach. At the same time it does not suit for all Indian languages. Using computer the text processing of any kind is possible with the help of software and hardware. Specially in case of forming conjuncts the shapes of the characters vary, these various shapes can be provided on computers by software. On the other hand the output on the typewriter is not up to the mark. Many a times' simultaneous use of English along with Indian script is required. This usage is made possible by standardizing the codes.

Indian scripts have tremendous applications in day-today life. These applications include, Word processing, Database, DTP, Teleprinting, speech, OCR etc. Once the characteristics of the scripts are known, making use of them for any of this application is possible. Standard code chart is designed for dedicated applications as well as with English. For any of this use, once the sequence of characters is known to form words or conjuncts, they can be sent to or received from the device and formation of exact word is done by software. This way device communication is normally, only the receive and sending device interprets the sequence.

2.6 Fonts for Indian languages/scripts

In simple terms, a font provides facility for displaying a set of symbols through well-defined shapes for each symbol. The symbol is a generic concept and the font is an instance of specific representation of a set of symbols. Traditionally, the symbols mentioned here have been the letters of the alphabet in a particular language along with punctuation marks and special characters. Fonts used to be created by craftsmen and

artists during the days of printing machines that used movable type faces. Today, fonts are created by artists and designers who work with computer based tools.

Inside a font the specific shape for a symbol is described either in terms of a digital image through bit maps or in terms of a filled outline. The former is called a bitmapped font and the latter, an outline font.

Outline fonts are increasingly being used on account of their scalability. The descriptions result in a pictorial representation or shape for each symbol, which is referred to as a glyph. Most fonts have a provision for describing upto 256 different glyphs, though in practice only about 200 may be present.

Each glyph in the font is specified by a name or an integer that stands for the location of the glyph in the set of 256. In the fonts for the Roman alphabet, it is common to locate the glyph for a letter in the place corresponding to the ASCII code of the letter. Thus, the glyph for "capital b" i.e., "B" will be in the sixty sixth position (the locations are numbered from zero). Most of the frequently used glyphs are seen in the first 128 locations of the font. The second half, known as the upper ASCII, usually contains special symbols, which are required in printed text to indicate phonetic aspects of the letter or a reference symbol for footnotes etc..

In most computer systems, a provision is available to display text using a font that may be selected by the user. The text to be displayed is represented through the ASCII codes of the characters to be shown. These codes may span the range 32-126, the usual set of values for the letters of the alphabet, or the range 160-254 normally reserved for special symbols.

There is no specific recommendation available on what symbols should get displayed via the upper ASCII range though the International Standards organization has recommended that the glyphs for some of the languages of Europe and the Middle east be assigned these locations. the term character set is often used to refer to the set of numeric codes assigned to the letters of the alphabet of a language. Thus, for a specified language, the code assigned to a letter of the alphabet will be the same in all computers so that application programs may recognize the letter from its internal code. If the glyph location for that letter also coincides with this code, then a one to one relationship exists between

the code for a character and its glyph location in the font. For most European languages, one letter invariably gets represented through one glyph.

The fixing of glyph locations for the letter has the most important advantage that the text to be displayed may be shown using many different fonts. This is precisely the idea behind wordprocessors permitting selected text to be displayed in a font chosen by the user. As of today, fonts for most of the languages of the world are limited to 8 bit codes for specifying the glyph positions. Almost all the languages of Asia, Japan, China and Korea cannot be specified through 8 bit codes for their letters as there are far too many of them. The Japanese character set includes some 24000 symbols while most of the scripts of India provide for as many as 12000-14000 individually differing aksharas. Later we will see how the aksharas of Indian languages may still be handled using 8 bit fonts, i.e., fonts supporting only upto 256 glyphs.

Chapter 3

Characters and Fonts Encoding for Indian languages.

3.1 Concept of Character Encoding

A document on the Internet can be written in various languages. The character encoding of a document represents a subset of the characters in the document. There are several character encoding that contain characters from a few scripts for example, ASCII (supports the characters in English), ISO-8859-1 (encodes most Western European languages), ISO-8859-5 (supports Cyrillic), SHIFT JIS (a Japanese encoding). The Unicode character encoding is a 16-bit character-encoding scheme, which can encode more than 65,000 characters. UTF-8 (Universal Character Set Transformation Format) encoding scheme is useful if English is used with other languages in the same document. When a document is transferred on the Web, the User-Agent on the other side should be informed what the character encoding of this document is.

3.1.1 ASCII

ASCII, The American Standard Code for Information Interchange is a standard seven-bit code that was proposed by ANSI in 1963, and finalized in 1968. Other sources also credit much of the work on ASCII to work done in 1965 by Robert W. Bemer. ASCII was established to achieve compatibility between various types of data processing equipment. Later-day standards that document ASCII include ISO-14962-1997 and ANSI-X3.4-1986(R1997).

ASCII is the common code for microcomputer equipment. The standard ASCII character set consists of 128 decimal numbers ranging from zero through 127 assigned to letters, numbers, punctuation marks and the most common special characters. The extended ASCII Character Set also consists of 128 decimal numbers and ranges from 128 through 255 representing additional special, mathematical, graphic, and foreign characters. Email, gopher, ftp use the 7-bit ASCII (American Standard Code for Information Interchange) standard, allowing for 128 characters. ASCII codes 0 to 31 and 127 are control characters

that are non-printable, whereas the other codes represent printable characters. The codes 32 to 126 represent characters of the Roman alphabet mapped to the standard qwerty keyboard. ASCII characters are each represented by one byte according to a standard code. Use of ASCII for information interchange has been severely limiting for languages written in non-roman scripts.

3.1.2 ISCII

The Department of Electronics came up with a standard for Indian code chart. The Indian Script Code for Information Interchange, ISCII[3], specifies a 7-bit code table, which can be used in 7 or 8-bit ISO compatible environment. It allows English and Indian script alphabets to be used simultaneously. ISCII retains the ASCII character set in the lower half and provides the Indian script character set in the upper 96 characters in the 8-bit code. Some of the main features of ISCII are as follows:

- ISCII character set is a superset of all the characters required in the 10 Brahmi based languages.
- The ISCII code contains only the basic alphabet required by Indian scripts.

Eight-bit ISCII code

ISCII-8 is an 8-bit encoding standardized by DOE in 1986. The lower 128 characters contains the ASCII character set, while the upper half of the table is used for Indian script code. The first two columns in the upper half are reserved for control characters as per the recommendation of the ISO(International Standards Organization). This encoding allows free mixing of Roman characters with Indian scripts.

Seven-bit ISCII code

Seven bit ISCII code is recommended for those computers and packages which do not allow the use of 8-bit codes. In 7-bit coding schemes, 128 positions are available for representing all characters of the script. In this encoding scheme, control codes of ASCII are retained and all other codes are used to represent Indian scripts. The disadvantage of this code is that Roman scripts cannot be mixed with Indian scripts.

3.1.3 Unicode

Unicode is a 16-bit character encoding that allows 65,535 characters compared to the 256 characters provided by 8-bit ASCII. The Unicode Character Standard is designed to support the interchange, processing and display of texts in various languages of the modern world. Unicode is the first plane of ISO-10646, which is also called BMP (Basic Multilingual Plane) or Plane Zero. ISO-10646 is a 32-bit encoding. It is divided into 32,000 planes, each of 64,000 character capacity. This permits 2,080 million characters. This form is also called UCS-4, Universal Character Set 4-bytes. Only the first plane of UCS-4 encoding, Unicode, is in use. The Devanagari block of the Unicode standard is based on ISCII-1988. The Devanagari characters are encoded in the range U+0900--U+097F.

3.1.4 UTF-8

Character encoding standards define not only the identity of each character and its numeric value, or code position, but also how this value is represented in bits. The Unicode standard endorses two forms that correspond to ISO 10646 transformation formats, UTF-8 (Universal Character Set Transformation Format)[4] and UTF-16. UTF-8 is a way of transforming all Unicode characters into a variable length encoding of bytes. It has the advantages that the Unicode characters corresponding to the familiar ASCII set end up having the same byte values as ASCII, and that Unicode characters transformed into UTF-8 can be used with much existing software without extensive software rewrites. Any Unicode character expressed in the 16-bit form can be converted to the UTF-8 form and back without loss of information. Some salient features of the UTF-8 encoding are as follows:

- All UCS characters $\leq 0x7f$ are encoded as a multibyte sequence consisting only of bytes in the range $0x80$ to $0xfd$.
- The lexicographic sorting order of UCS-4 strings is preserved.
- All possible 2^{31} UCS codes can be encoded using UTF-8.
- The bytes $0xfe$ and $0xff$ are never used in the UTF-8 encoding.
- UTF-8 encoded UCS characters can be upto six bytes long.

- The first byte of a multibyte sequence which represents a single non-ASCII UCS character is always in the range 0xc0 to 0xfd and indicates how long this multibyte sequence is.

3.2 Concept of Font Encoding.

The encoding of a font specifies the mapping from character codes (an integer, typically between 0 and 255) to the characters. There are no particular standards for encoding fonts and almost every vendor comes up with an encoding scheme that is convenient for use in their software. Though character encoding is different from font encoding, the font encoding could be the same as the character encoding and this would greatly reduce the standardization problem of font encoding.

Font encoding basically refers to the set of glyph locations recognized by a computer system. The glyphs need not bear any relationship to the letter (Roman) of the alphabet associated with the glyph location. The font designer can therefore place a glyph in any desired location but refer to the glyph using the code assigned to that glyph location (or equivalently, the name assigned to that glyph).

The concept of font encoding allows us to generate displays of text strings in many different languages by designing fonts which contain the glyphs corresponding to their alphabet. The text to be displayed is represented as a series of eight bit characters and for all practical purposes, these may be reckoned as a string of ASCII codes. The computer system takes each code and displays the glyph associated with it. The glyphs may be viewed as the building blocks for the letter to be displayed where, by placing the glyphs one after another, the required display is generated. Fonts also incorporate a feature whereby some of the glyphs may be defined to have zero width even though they extend over a horizontal range. Thus when the system places a zero width glyph next to another, the two are superimposed and thus permit more complex shapes to be generated, such as accented letters. Zero width glyphs are very important for Indian language fonts.

The location of a glyph within a font that caters to non-Roman letters is pretty much arbitrary though designers tend to follow one or more encoding standards. Different encoding methods are in use today where each encoding has standardized the locations where glyphs may be placed. Unfortunately, these locations are not uniform

across different encoding. This is a consequence of different vendors or independent groups choosing to support their own encoding standard in their applications (or the Operating System). Over the years, the following encoding standards have become popular.

1. The standard Latin character set as per ISO-8859-1

This encoding is recognized under most Operating Systems. The standard provides for about 190 glyphs.

2. Windows specific Latin character set known as Latin-1252

This encoding supports a dozen or more glyphs beyond the number supported under 8859-1. This encoding is the most common choice under Microsoft Windows, in respect of non-Roman fonts.

3. Macintosh Encoding.

The Mac encoding is similar to the windows 1252 but differs in a few specific glyph locations.

4. Encoding supported under PostScript.

PostScript is an independent approach to generating printed documents based on a Graphic description language developed at Adobe Systems, USA. There are two or three different encoding supported under PostScript but it is possible to have user specified fonts with arbitrary encoding.

More glyphs in a font means that the font will permit display of a more comprehensive set of letters and characters. However due to variations in text processing across different computers, only the first of the above mentioned standards is really usable across different computers.

Fonts are inherently proprietary in nature and tend to be incompatible across computer systems as well as applications. Today, it appears that very few non_Roman fonts are available for practical use which are supported under all the important platforms. This has imposed fairly severe restrictions in respect of web displays of Indian Language text on account of the totally arbitrary approach to designing the fonts. True, these fonts were not designed with the idea of text processing but more for getting

good printouts. However when we use the same for web pages, we run into many incompatibilities.

In editors and word processors, the internal representation of text is the ASCII code of the letter displayed and these codes happen to be the same as the numeric code assigned to the glyph locations containing the letters. When it comes to fonts for Indian languages, the display has to be built up with more than one glyph for many aksharas and hence the internal representation of the aksharas is purely a function of where the glyphs for the akshara are located within the font. Thus one faces the problem that the stored text is not in a format that can be viewed on different computer systems because the encoding standard may not be supported in each system. Also, glyph codes are the choice of the font designer and will bear no relationship to the ordering of the aksharas in our scripts. Thus linguistic processing of the stored text is a formidable task, being font dependent even for a given script.

The crux of the problem in respect of Indian language fonts is best illustrated through an example. Xdvng is a popular font for Devanagari that follows the ISI-8859-1 encoding. Sanskrit_1.2 is a beautiful font for Devanagari which is available only for the Windows platform. Shown below are the glyphs present in each of the fonts. Sanskrit_1.2 provides for many more glyphs and includes some Vedic symbols as well.

Clearly the ASCII strings required to display Devanagari text differ for each font. Also there are some aksharas which may be displayed in Sanskrit_1.2 but not in Xdvng. One would also notice that the number of glyphs required to display an akshara depend on the akshara (or samyuktakshara) and thus a variable number of bytes is required to represent each akshara. This is in contrast with the western alphabet where one letter is displayed through just one glyph.

The phonetic aspect of Indian languages generally compact a lot of phonetic information in a displayed akshara. A complex conjunct may contain as many as four consonants and a vowel but may be displayed as a unique shape with just one specially designed glyph. Thus, no matter how best we attempt to design fonts for our languages, we are beset with the problem of having to deal with text in terms of multiple byte representations for the aksharas. Text processing with variable number of bytes for each unit of information is a nightmarish proposition and it is precisely for this reason that we

have not seen general applications supporting Indian language user interfaces which work in a universal manner on all computer systems.

The solution to one akshara one glyph for Indian languages lies in the adoption of sixteen bit codes for the aksharas. We will then have a set of nearly 14000 different codes for our languages where each code is exactly 16 bits. Also if we can have fonts developed to accommodate that many glyphs (as is the case with Japanese characters), then a one to one correspondence between the akshara and its glyph will be possible. The only difficulty today is that no computer system will look at a 16 bit character code, index into a 16 bit font and display the glyph. Unicode, though claimed to be a sixteen bit code, is really an eight bit code for each language and is therefore not suited to this requirement. In respect of Chinese, Japanese and Korean, Unicode does indeed support a fixed 16 bit code for each character (about 24000 of them) but the problem of large numbers of aksharas in our languages has never been properly addressed by Unicode.

The encoding of a font specifies the mapping from character codes (an integer, typically between 0 and 255) to the characters. There are no particular standards for encoding fonts and almost every vendor comes up with an encoding scheme that is convenient for use in their software. Though character encoding is different from font encoding, the font encoding could be the same as the character encoding and this would greatly reduce the standardization problem of font encoding.

3.2.1 Problems faced in designing fonts for Indian languages.

In respect of Indian language text display and processing, we are confronted with the following issues. There is no simple solution to the problems faced.

1. We cannot have standard fonts for each script as the number of glyphs required to form the aksharas properly is more than 256. Any attempt at standardization will necessarily have to compromise on the approach to displaying the conjuncts. Worse still, this will have to be done individually for each script.

It may be possible to agree on a compromise set of glyphs for each script and request font designers to place them at specified locations in the fonts. This way, some uniformity may be possible in displaying the text for that script. However, this may not

be feasible in practice since printing requirements are fairly stringent on the conjuncts and require that many be included.

2. An eight bit representation of the consonants and vowels may be feasible but one will have to live with multibyte codes for the aksharas. ISCII was an early attempt at achieving some uniformity but this was not sustainable across all the Indian languages.
3. We need to support automatic transliteration to permit language independent information to be read in any script. Here one is confronted with the identification of a global set of aksharas.
4. A one to one mapping between an akshara and a glyph is ruled out if eight bit fonts are used. As of now, 16 bit fonts are not handled properly in most computer systems.
5. There is no clue to how many aksharas are actually needed in a language, for many new samyuktaksharas may be formed. The set of 14000 may eventually increase.
6. It is not merely the aksharas we must display. For educational needs we also must display the symbols for the matras along with special letters and punctuation. These should also be coded into the character set and explicit glyphs for these must be provided. Due to the variable physical width of the aksharas themselves, more than one representation for a matra may be needed (see the glyphs of the Xdvng font).

3.2.2 Fonts as used in web pages and web displays

Indian language text may be conveniently displayed on web pages using the html based approach. While it is indeed possible to display our texts this way, certain interesting problem crop up when transparent viewing across different web browsers is required. It is true that the html standard does allow user specified fonts to be used for the display but there is no guarantee that all web browsers will support the particular encoding specified in a font. Most web browsers can be told the encoding to be used for the font which will display the downloaded page, automatic switch over to the encoding applicable to individually specified fonts within the downloaded page is never guaranteed.

It has also been observed that Dynamic fonts, a very interesting and useful concept which permits the fonts used in a web page to be sent along with the page, are rendered properly only if they conform to the ISO8859-1 Latin encoding. As of now Dynamic fonts generation tools work only with true type fonts. Many Indian language fonts in TrueType format are not rendered properly on Unix systems as these fonts for Indian languages are encoded according to windows-1252 encoding which permits several more glyphs to be accommodated in the font compared to iso-8859-1. In fact, Sanskrit_1.2, a truly high quality Devanagari font, does not get rendered properly when sent as a Dynamic font. This is a very useful font as it supports Vedic symbols as well but it is usable only under Windows-95. The same applies to a Telugu font called Pothana.

It appears that as of today, the safest approach to displaying Indian language text on web pages so as to be viewed from almost all the browsers, is to restrict the font used to an ISO-8859-1 encoding. If Java based applications are considered, then it is even more important that we stick to this encoding.

Such a restriction always goes against the wishes of font designers who would like to accommodate as many glyphs as possible to allow a richer set of aksharas to be rendered. Among the freely available fonts for Indian languages, very few seem to conform to this requirement of 8859-1 coding. This may be a consequence of the fact that font design tools are more easily available for the MSWindows platforms and thus most designers' end up producing TrueType fonts.

Chapter 4

Design and Implementation

The limitations one meet while communicating in various languages through the Internet is that due to lack of sufficient standards and protocols. Presently language content in UTF-8 encoding only have been possible with the available standards. Indeed, UTF-8 is a safe way of encoding for transmission across various networks. Moreover, most of the computers have the "qwerty" keyboard, using which typing in different languages is not possible unless there is a keyboard driver that allows the user to do this. Users have to depend on Latin-1 to fill forms. This restricts Multilingual features of the Web in a big way. Our design provides a way to get around this problem of feedback in Latin-1 by providing a Java applet that allows users to type in Indian languages. The applet can then communicate with the server and transfer the information provided by the user.

4.1 A Solution through Java Applet

As a solution to have text processing facility in Indian language, we have provided a Java applet, that acts as a keyboard driver and allows users to type in Indian languages. The language in which the user can type and the font dependent information can be provided to the applet through the various configuration files.

The advantage of a Java applet is that it is platform and operating system independent and it can be easily downloaded across the network and interpreted by the browser. Further applets written in Java do not pose any security threats to the client system. Java applets can be easily embedded in HTML documents. They can be used to create smart forms, that are interactive, and can do input verification at the client side, rather than verifying on the server, which can save bandwidth.

4.2 Configuration Files

The Java applet requires the following configuration files while startup. The place where the applet can find these files on the server is specified in the !PARAM? tag in the

HTML file that embeds the Java applet. The following have to be specified in the HTML file:

- ISCIIFILE, the file containing the character encoding information
- KEYBOARDFILE, the file containing information about the layout of various keys.
- FONTTABLEFILE, the file that contains information on the conjuncts in the language.
- FONTFILE, the file that contains information of the glyph positioning in the corresponding font that is used.
- UNICODFILE, is the file that contains the mapping between Unicode and ISCII.

The internal representation of characters are in ISCII form for processing, but in Unicode format when stored or when sent across the network.

4.2.1 Character Encoding Format

This file is provided so that the other configuration files can be easily written and interpreted. This file has the format:

STRING_REPRESENTATION_OF_CHARACTER CHARACTER_CODE



The character code is an integer. It can be an 8-bit character code or a 16-bit character code. As this string representation of characters is used in all the other files, this helps in using any character encoding the user prefers. The file name should be set as the value for ISCIIFILE parameter. In this file the string representation of a character as well as the character code are one word each. The string representation is not standard, but it is used internally by the applet in reading other configuration files.

4.2.2 Keyboard Map

The layout of various keys on the keyboard is specified in the "keyboard map" file. This file name is specified in the HTML file, which embeds the Java applet as a parameter using the !PARAM? tag. By changing the file, the user can give a keyboard layout of his choice. This file should be written in the following format:

ASCII_value_of_the_key A_set_of_ISCII_codes_in_string_format

TH-7860

The set of ISCII codes represent the codes corresponding to the character in that layout. If a key corresponds to more than one ISCII code, then the codes are written in the same line separated by whitespace. This file should be specified as the value of the parameter KEYBOARDFILE. In this file, ASCII value of a key is a single word, but the ISCII codes can be any number of codes mapped to that key.

Conjuncts

Conjuncts are clusters of upto four consonants without the intervening implicit vowels in Indian scripts. The shape of these conjuncts can differ from those of the constituent characters. Different Indian scripts have different conjuncts and hence have to be defined in the configuration file, so as to make the applet language independent. The format of the conjunct file is as follows:

```
STREAM_OF_ISCII_CHARACTERS      INTERMEDIATE_GLYPH_CODE
```

The ISCII characters are separated by whitespace. The intermediate glyph code is an integer which is used to represent each possible character that can be generated from the font. This file can be specified as the parameter FONTTABLEFILE. The intermediate glyph code is a unique number and is the last word of each line. The rest of the words correspond to the ISCII characters that are represented by the glyph code. The glyph code is not a standard code, but used internally by the applet.

4.2.3 Font Information

The font information file specified in the parameter FONTFILE, gives the mapping between the intermediate glyph code and the glyph code used in the font. This file is dependent on the font that is used as every font has its own encoding scheme. This file also gives us the information about how many glyphs in the font correspond to a particular character in the alphabet. This file should be in the format:

```
INTERMEDIATE_GLYPH_CODE  
GLYPH_CODES_CORRESPONDING_TO_THE_CHARACTER
```

As in the other configuration files, the glyph codes corresponding to the character are separated by whitespace. All possible glyph sequences have unique intermediate glyph number. These intermediate glyph numbers can map any number of glyphs in the font.

4.3 Devanagari Text Area

JDK1.2 provides `JTextComponent` class for text editing features. The `TextComponent` class is a super class of a set of classes. The most commonly used components for editing purposes are the `JTextField` and `JTextArea`. The `JTextComponent` provides these customizable features for all its descendents:

- A separate model, known as the document, to manage the component's content.
- A separate view, which is in charge of displaying the component on the screen.
- A separate controller, known as an editor kit, that can read and write text and that implements editing capabilities with actions.
- Customizable keymaps and key bindings.
- Support for infinite undo and redo.
- Pluggable caret and support for caret change listeners.

4.3.1 Documents

Like all Swing components, a text component separates its data (known as model) from its view of the data. A text component's model is known as a document. A document provides these services for a text component:

- A document stores the textual content, which can represent any logical text structure, such as paragraphs, text runs that share styles, and so on.
- Provides support for editing the text through the `remove` and `insertString` methods.
- Notifies document listeners and undoable edit listeners of changes to the text.

The `DevTextArea`, designed by us, is a subclass of the `JTextArea`, which is inherited from the `JTextComponent` class. The `DevTextArea` has its own document, which is used to display the variable number of glyphs associated with each character in Devanagari.

4.3.2 Keyboard to ISCII encoding

The DevTextArea uses the inscript keyboard layout for the input of characters. The code of the typed key is converted into a corresponding ISCII code. The following subsections describe the process in detail.

4.3.3 Keyboard Map

The "keyboard map" specifies the mapping between the "qwerty" keyboard and the inscript keyboard. The "keyboard map" file consists of the key codes and the corresponding ISCII characters mapped to the key. Whenever a key is pressed the corresponding ISCII codes for the key are generated and this serves as the input to the routine which generates the display codes. As a key pressed generates a set of ISCII codes which is independent of the font code, it is possible to generate the same character to be displayed for all the key sequences required to generate a particular character. We use the inscript keyboard layout, in which the vowels and the consonants of the Indian language alphabet are arranged on the left and right side of the keyboard respectively.

4.3.4 ISCII

The "iscii" file contains the mapping of symbols to the 8-bit ISCII codes. The appendix lists the table of ISCII codes. This mapping decreases the complexity of other files as other configuration files are written using these symbols and makes the configuration files readable.

4.4 Formation of Devanagari Characters

Characters in Devanagari are represented as a series of ISCII codes. To obtain a character in Devanagari, the constituent ISCII characters are combined based on the information in the "font table" file, which specifies the set of ISCII characters that correspond to the display code.

4.4.1 Generation of Characters

We construct a hash table, in which a set of ISCII characters are mapped to a display code. If the input pattern consists of an ISCII character sequence which is a key

in the hashtable, the corresponding value is stored in the display buffer. The display buffer thus consists of a set of display codes corresponding to a series of glyphs.

4.4.2 Generating Glyphs from Intermediate Display Sequence

All possible characters in Devanagari are given a unique numbering used for internal representation. This numbering is different from the font code. The display code numbering corresponds to logical characters in Devanagari. Once the display code is obtained from the ISCII code, the characters to be displayed are generated using the "FONTFILE" file, which contains the mapping between the display code and the font code. The insertString method of DevTextArea's document model, does the conversion between display code to font code. Thus characters which comprise of one or more glyphs are generated and these glyphs are displayed in the DevTextArea.

4.5 Position of Matras

Matras are vowel signs that are added to consonants to indicate a vowel sound other than the implicit one. All matras in Devanagari, except the Matra E are positioned after the consonant. The Matra E is positioned before the consonant. Because of the phonetic nature of ISCII encoding, the code of Matras appear after the code of the consonants.

4.6 Caret Positioning

Whenever a character is added in the text area, the caret position has to be changed to show the user a logical position where the next character would be added. As a display code might represent more than one glyph, the caret has to be moved as many places as the number of glyphs inserted, as otherwise the caret may be positioned between the glyphs constituting to the character. The "fontinfo" file contains a mapping between display code and font code. Thus from this file we can find out the number of glyphs corresponding to a particular display code. With this information the caret can be positioned at the appropriate place. Caret Movement In English, there is a one to one correspondence between the input character and the symbol displayed. So movement of the caret corresponds to positioning the caret according to the number of glyphs present. However, in Indian scripts many input characters may combine to form one display

symbol. When this kind of input occurs, the caret is moved to the end of previous syllable when the user presses the left arrow key. Movement of the caret is done using the `setCaretPosition` of the `JComponent` class. The position of the caret is calculated and the `setCaretPosition` method is called with the appropriate caret position as its parameter. As a different number of ISCII characters may combine to produce a particular glyph, a separate buffer called the `isciiBuffer` maintains the ISCII string input through the keyboard. When the user moves in the text, the corresponding movement in the `isciiBuffer` also has to be obtained. This would result in the proper insertion of the characters as the user might want. The change in the caret position leads to a corresponding change in all the buffer positions.

4.7 Text Manipulation

The `DevTextArea` serves as a Hindi text editor. Characters can be inserted at the current caret position. The caret positions are demarcated by syllables. One can insert text before or after syllables but not in between the characters in the syllable. Characters cannot be inserted between the constituent characters of a syllable. To insert a character between the constituent characters of a syllable, one has to break the syllable by pressing the Backspace key by positioning the caret after the syllable and rewriting the word from that position.

Since we modify the underlying document of the `TextArea` after every key pressed, other features of any `JComponent` like scrolling can be easily supported by enclosing the `DevTextArea` in a `JScrollPane`.

Chapter 5

An Online Chat Application in Indian Languages

In this chapter, we have designed a Chat Applet in Java. The applet reads a log file stored on the server repeatedly at a given interval of time. This log file stores the last twenty chat submissions. We use a thread to read the log file every 8-10 seconds. Any new data in the log file is appended to the text in a TextArea. Chat data are sent to the server in a separate thread. Each person will be required to enter a name to chat with. When a person enters or leaves the Web pages with the chat applet on it, a message will be submitted to the server that informs others that he/she has either entered or exited the chat room. While it would be a trivial task to read from the log file in a Java Applet, sending data (to be appended) to the log file, using http, becomes a challenging task. Each line of the log file will represent one line submitted to chat. Each chat line received will be appended to the beginning of the log file so that the most recent submissions will appear at the top of the file.

5.1 Building the Chat Applet

Each time the Java Applet reads the log file, it will read the most recent chat lines first. The Applet will compare each line to the first line read in the previous reading. If these lines are different, then the chat log has been updated since the applet last read it. The Applet will continue to read the file until it either finds a match or reaches the end of log file. Each line read are stored in a temporary String, appending each line read to the beginning of the String. This will re-order the chat data with oldest chat first to most recent chat last. Then temporary String is appended to the end of the TextArea, where the users read the chat data.

5.1.1 Submitting a Chat

To submit a chat, we need to emulate the interaction that happens when a Web browser uses CGI to send data to a Web server. A Web browser can use GET or POST to send data to a Web server. A form in HTML is made where the GET method is used,

when a person presses the Submit button the browser would request the following or a similar URL.

```
http://www.myserver.com/cgi-bin/
```

5.1.2 Reading the Chat data

The POST method works almost the same way as the GET. The main difference is that the data is read from standard-in instead of the query string. It is almost always a good idea to use the POST method over the GET, since the GET method has limitations.

```
Sdate = &ctime(time);  
chop($date);  
Sdate = substr($date,4,12);
```

The lock file is simply a flag which indicates that someone has opened the log file. When the user is finished with the log file, he/she will delete the lock file. The idea behind this lock file is:

- It needs to be checked if a lock file already exists. If it exists, someone else is in the process of writing to the log file. So one needs to continue to check for its presence for 60 seconds.
- If the lock file disappears within the 60 second period, a new lock file is created so that others will know that the log is being used currently. When it is done with the log file, later the lock file has to be deleted using the `&release_file_lock` function so that others can use the log file.
- If the lock file does not disappear, the user will assume that something went wrong. If the Web server was rebooted while the lock file exists, for instance, it will reboot, in error, with a lock file existing. Sixty seconds is more than enough time to wait. After waiting this time period, one will take control of the lock file for himself.

A class called `SubmitToChatServer` has been created which implements `Runnable Thread` to submit the data. This object uses the standard `Start()` method to create and spawn the Thread. Objects of this class will be constructed with the following parameters:

- String text This represents the actual chat text to be sent to the chat server.

- Applet app – This is a handle to the applet which is needed to construct the URL in order to send the chatline, as well as to show status in the applet.
- String chatFileName – This specifies the name of the file on the Web Server which will store the chat log.

We extract this file name from a parameter listed in the HTML file, which loads the Applet. The reason we want to do this is so that we can use a single chat applet to open several chat rooms. The HTML page, which has the name of the chat log file embedded in it as a parameter, can be created on the fly.

The constructor will create a String, named completeMessage, by encoding the data that will be sent to the Server by using a utility method in the URLEncoder class, called encode(). We must separate each element with an & symbol and use the = sign to assign the value. The URLEncoder.encode() method will replace illegal characters, such as spaces, with characters that can be transmitted to the server without problems.

One method we need to call is setDoOutput(true). This provides the ability to send our chat data to the Web server. In order not to cache data, the setUseCaches(false) method is called on our URLConnection object. The Post method requires that the user specify what type of data he/she is sending and its length, in bytes. Since the data are being Posted, the Content-type and Content-length using the setRequestProperty() method need to be specified. The parameter sent as: eMessage.length()” uses a kind of hack to convert the returned int from completeMessage.length() into a String.

Once the URLConnection is set up like this, a DataOutputStream object can be created with an output stream from the URLConnection object. Using this DataOutputStream data are sent to the Web Server with the writeBytes() method.

The world sees the Java applet as a chat room. So we have chosen to create an applet object and a Panel object. The applet will simply act as a container for the Panel object. The Panel object will then be the meat of the project. The Chat class is derived from the applet Class. This class will be called by the HTML web page, which contain parameters that the applet will extract. The log file, which gets created and maintained by the Java Script, will reside on the Web server. In it the last ten chat submissions are stored in order of most recent first.

The `init()` method of the `Chat` class uses the ternary operator as a method to extract the parameters in the HTML file. The `init()` method also makes an instance of the `ChatPanel` object which will actually contain the chat engine. After setting the applet's layout to `BorderLayout`, the `ChatPanel` object is added to it in its `Center` position.

The `start()` and `stop()` methods just call the `ChatPanel` object's `start()` and `stop()` methods, respectively. This will be important not only for starting and stopping the chat thread, but for sending a message to the Web server that the client has entered/exited the chat room.

```
public void start(){
    chatScreen.start();
}
public void stop(){
    chatScreen.stop();
}
```

To construct this object, we will pass two parameters:

- A handle to the applet
- A String that will specify the file name of the chat log

The `start` method checks to see if the user has entered a name. If the user has not entered a name to chat with, it will bring the setup panel to the front and request focus for the `TextField`, which will be where the user will enter a name. If the user has entered a name, then a new `SubmitToChatServer` object is created with a message indicating that this person has entered the chat room. Each line we send to the chat server will work exactly in this way, spawning a separate thread with which to send the data. Next, we will start our chat engine by creating another thread for it, if one does not already exist, and starting it. This will, in turn, invoke this object's `run()` method in the thread

The `stop()` method will do a similar thing to what the `start()` method does. It will create another `SubmitToChatServer` object with a message indicating that the user has left the chat room and start the process of sending it in another thread. After doing so, it will stop the chat engine's thread so that it will not be constantly refreshing the screen.

```

public void stop(){
    if(runner != null){
        new SubmitToChatServer("*** "+identity+" Has
        Left This Chat Room. ***",app, chatFileName).start();
        runner.stop();
        runner = null;
    }
}

```

The run() method will be the heart of the chat engine. The first thing we do is request focus on the textField, which will be used for chat entry. In order to wrap the text, we need to figure out the width, in pixels, of the average character in a specific font. Once this is done, we go into an infinite while loop, which invokes a method called refresh() and pauses the thread for the length of time specified in refresh time.

Next, we create a DataInputStream object, which we will use to read its data. Since the log file stores its most recent data first, we will need to read each line and prepend it to the front of a String, named chatData, which will be used to store the new data with the oldest data first. The first line we read (Most Recent Chat) will be stored for the next time this method is called. Each line read will need to be processed by the processText() method. This method will perform the wrapping of text if necessary. We will keep doing this on a line by line basis, until we read a line that matched the first line read from the previous iteration of this method. When finished, we will append the temporary String to the chatArea TextArea and close the stream. To force the TextArea to scroll down to the bottom, we will select the last character of text contained within it

The processText() method is responsible for word wrapping within the TextArea. It takes the nprocessed text in as a parameter and returns the processed text as a String. If the text contains fewer characters than the average characters per line, then we will simply return the text with two new-line characters appended to it. If the text contains more characters than the average characters per line, then we examine the first group of characters that fit on a line. We count backwards until we find the last instance of a space. Here, we will insert a new-line character and repeat the process on

the remaining text after we pre-pend the person's name, which we extract from the text, to the beginning of the remaining text. When finished, we return the completed processed text with two new-line characters appended to the end.

```
new SubmitToChatServer(identity+" >:
"--chatLine.getText(),app,chatFileName).start();
chatLine.setText("");
}
```

To finish off this class, we have written a number of methods. These include:

```
setRefreshTime()
setIdentity()
getIdentity()
setChatLogName()
```

There may be a number of things one wish to do to optimize this Chat applet. For instance, one may want to move the refresh time up or down depending on his/her chat traffic. However it will be refreshing at longer or shorter intervals and each refresh will read the current log file, which stores up to twenty lines of chat. If the user has heavy traffic in his/her chat room, more than twenty lines may have been submitted during that time period. The user may want to alter the Java Script to store more lines; however, twenty seem to be more than adequate in most situations.

5.2 Adding Indian Language Feature to the Chat Application

The problems in having a facility for online chat in Hindi can be summarized as follows:

- Different fonts have different encoding schemes.
- Most of the existing UNIX systems use 7-bit encoding schemes to transfer information through e-mail.
- Most systems come with the "qwerty" keyboard, using which typing messages in Hindi is not possible.
- Even if the font is present on the system, there is usually no regular mapping between keyboard and font code. So getting the appropriate characters would be difficult.

- If we use the font code for encoding characters, it would not be possible to encode all the characters using the 7-bit code.

5.3 Sending Chat data in Hindi

In this section we discuss about how to send Hindi and English texts dynamically using the existing resources.

5.3.1 Typing the Chat data in Hindi

We use the DevTextArea, which serves the purpose of an editor, to type the messages in Hindi. As we have discussed earlier, this editor uses the Inscript keyboard layout, in which the consonants are on the right side of the keyboard and the vowels on the left side. Hence it is very convenient to type the messages in Hindi and English using this editor. The DevTextArea represents the message typed as a stream of ISCII characters. These ISCII characters are converted to Unicode characters. The Unicode characters are sent over the network by encoding the characters in UTF-8.

5.3.2 Encoding the Chat data in UTF-8

Characters in Unicode are of 2-bytes length. Hence, they cannot be sent over the network directly. So they are encoded into the UTF-8 encoding, in which the Unicode characters are represented as a set of two separate bytes. The StringEncoder class has a method, `unicodeToUTF8`, which converts the given Unicode string into a set of UTF-8 bytes. These bytes are ready to be sent over the network.

5.3.3 SMTP

Our Chat client uses the Simple Mail Transfer Protocol, SMTP, to send messages. A client delivers a message by opening socket connection to port 4455 on the server. Once the connection is established the client sends a series of commands, delivers the message and disconnects.

All commands and responses to and from the SMTP server are terminated by CRLF. For every command sent by the client, the server responds with a text response, that starts with a 3-digit code. The server response codes are categorized in Table 5.1

Response Code	Explanation
200-299	Successful command execution

300-399	Command was initially successful but it needs more information to complete
400-499	There was an error on the server(server down, file system full)
500-599	Indicate an error on the client side (Invalid password, unknown command)

Table 5.1: Response codes in SMTP

The sequence of commands sent from the client to the server is:

- HELO host name - a simple greeting from the client to the server. The host name field is optional, but the sendmessage programs require the host name field.
- MESSAGE FROM: sender's address - identifies the user account which is sending this message. No check is performed by the server on to who is sending this message. So it is virtually possible to send a message as anyone in the world
- RCPT TO: receiver's address - identifies the address where this message will be sent.

The SMTPSession class implements a session with the SMTP server. It is used by our applet to send the message. The sendMessage method essentially opens a socket on the SMTP port and writes the headers and other information essential for initiating the SMTP session. Then the body of the message is written on the socket. Any error occurred during the sending of message is reported by the SMTPSession class.

5.4 Receiving Chat Data typed in Hindi and English dynamically.

Our Chat client uses POP3 to fetch the message from the server. So the applet should be fetched from a webserver, that also acts as the POP3 server for the local network. The following sections describe in detail how the message is being fetched and displayed to the user.

5.4.1 POP3

Post Office Protocol version 3, POP3, is a popular protocol used to remotely access messages on the Internet. By using the POP3 protocol, the user can read his/her messages without having to copy on the local machine. POP3 responses start with a "+" for a successful command and a "-" in the case of an error. POP3 responses could span

only a single line or many lines. If the response is a multiple line response, it is terminated with a line containing a single ".". After connecting to a POP3 server, you have to send a USER user name command, followed by a PASS command giving the user's password.

The POP3Session class acts as a wrapper to the application, whereby it takes care of the connection establishment, reading a message, converting the raw sequence of characters into a Message, the data structure that represents a message.

5.4.2 Decoding the Chat Data

The message fetched using the RETR command contains the headers as well as the body of the message. These headers are stripped from the message using find the method name in the Message class. The body member of the Message class contains the body of the message after the headers are stripped off. The body of the message is what we are interested in as it consists UTF-8 encoded Unicode characters, which represents the Hindi as well as English string.

We convert the UTF-8 encoded characters into Unicode using the UTF8ToUnicode method of the StringConverter class. This sequence of Unicode characters is converted into ISCII using UnicodeToIscii method of the StringConverter class.

5.4.3 Displaying the Chat Data

The method setISCIIString of DevTextArea converts the given sequence of ISCII codes into a sequence of display codes, by using the longest matching sequence of conjuncts as specified in the "font" file. The display code corresponds to a set of font codes. When a particular display code is inserted into the text area, the underlying document of the DevTextArea replaces this display code with a sequence of font codes. Thus the text displayed in the message consists of a sequence of Hindi and English characters.

Due to applet security restrictions, which state that an applet can make network connections only to the host from which it is downloaded, it is required that the server from which the applet is downloaded has to be the SMTP server as well as the POP3 server.

5.4.4 Online Chat in other Indian Languages

The application can be extended to display any other Indian language as all the Indian languages have a similar structure. To send mails in other Indian languages, one has to ensure the following things:

- The appropriate Indian language font has to be installed on the client machine.
- The keyboard layout of the particular language has to be specified in the "keyboard_map" file.
- Though the ISCII codes for all Indian languages are the same, the characters have different Unicode values. The configuration file which contains the mapping table between ISCII and Unicode has to be changed appropriately.
- The conjuncts in the appropriate Indian language have to be mentioned in the configuration file which has information about conjuncts.

Chapter 6

Conclusions and Future Work

In this thesis, we have presented a Java applet that can be embedded into a HTML page, which is used to get user input in Indian languages. The solution is secure, portable and transparent to the end user. The solution was implemented for Devanagari script together with English. Devanagari forms the basis for documents in Hindi, Marathi and Sanskrit. Due to constraint of time and resources the solution could not be implemented for other Indian languages as it was intended. This solution can be extended to other languages just by providing the configuration files required at startup. We have also implemented a Chat client that allows users to transparently send and receive chat messages in Hindi and English.

6.1 Limitations

Our solution has the following limitations:

- The solution requires the Java plugin 1.2 to be installed on the client machine.
- At present, only one Indian language with English can be used in the applet.
- Moreover, the applet does not allow dynamic change of fonts in the input. It requires the font to be installed on the client machine.
- The startup time of the applet is quite high. This is mainly because of the high startup time of Java applets.
- The Chat applet requires the Web server to be support SMTP as well as POP3 protocols, as it cannot make network connections to other hosts in the network.
- The Chat applet also requires addresses to be typed using Latin-1 encoding.

6.2 Future Work

Our applet supports only one Indian language together with English at a time. This can be extended to support multiple Indian languages.

- The applet can be extended to support more than one font in the same input area.
- It can be extended to support styled text as input.
- The Chat applet can be used to exploit the security model provided by Java and connect to any server on the network. This can be done by changing the default security model provided by Java.
- The applet can be enhanced to allow users to enter URLs in Hindi. This can be done by providing a small dictionary interface between the ISCII code for the URL and the absolute URL in Latin-1.

Bibliography

1. iLeap - Intelligent Internet Ready Indian Language Word Processor
<http://www.cdac.org/gist/ileap1.html>
2. Gist Multi-lingual Card user's guide, Quark Computers Pvt. Ltd., C-1, Sarvodaya Nagar, Kanpur, 1996
3. Indian Script Code for Information Interchange - ISCII, IS 13194 : 1991, Bureau of Indian Standards, Manak Bhavan, 9 Bahadur Shah Zafar Marg, New Delhi 110002.
4. F. Yergeau, UTF-8, a transformation format of Unicode and ISO 10646, Request for Comments: 2044, October 1996
5. The Unicode Standard, Version 2.0, The Unicode Consortium, Addison-Wesley Developers Press, Chapter 6 pp33-34, Chapter 7 pp72-74
6. The Java 2 Platform API Specification
<http://java.sun.com/products/jdk/1.2/docs/api/index.html>
7. The Java Tutorial
<http://java.sun.com/docs/books/tutorial/index.html>
8. Jonathan B. Postel, Simple Mail Transfer Protocol, Request for Comments: 821, August 1982
9. J. Myers, M. Rose, Post Office Protocol - Version 3, Request for Comments: 1725, November 1994
10. Mark Wutka, et. al. Hacking Java: The Java Professional's Resource Kit, Macmillan Publications, 1997