HKU CSIS Tech Report TR-97-02

A framework for the support of multilingual computing environments

葉志立 Yip, Chi Lap <*clyip@cs.hku.hk*> Ben Kao <*kao@cs.hku.hk*>
David Cheung <*dcheung@cs.hku.hk*>
Department of Computer Science, The University of Hong Kong *

Abstract

The issue of multiple natural language support in operating systems and application programs has appeared and reappeared under many different headings. "Internationalization", "localization", "double byte character set (DBCS) support", "character composition support", and "language toolkits" are only some of them. Yet, the issue concerns so many areas that most existing solutions address only part of the problem.

This paper describes an integrated approach which addresses the problem from a system point of view. We outline a framework that unifies the different approaches in supporting multilingual input, output and user interface. The idea is to design a specialized distributed system database (MLDB) and a software library (MLLib) based on which a framework is built. Specifically, MLDB is a system-level database specialized in handling various information about multilingual environments. This includes the coded character set used, input methods available, and text messages. MLLib, on the other hand, abstracts the details of multilingual text handling and provides programmers an application programming interface (API). With MLLib, programmers can easily write application programs that interact with users in more than one natural language at the same time. We argue that these components together will provide a system that is user-, programmer- and system administrator-friendly.

Keywords: Multilingual Computing, Natural Language Support, Network Computing.

1 Natural language support

With computers networked together worldwide, transmission and distribution of data become easy. With just a few keystrokes, one can obtain a document from the other side of the globe. The network can thus be considered as a global base of data. Among the

*Pokfulam Road, Hong Kong. Phone: (852) 2859 2180; Fax: (852) 2859 8447

different types of data a computer can handle, text is perhaps the most important type. Support of text is thus a fundamental problem in most computing environments.

To contribute to our knowledge, mere data have to be transformed into information first. With a worldwide contribution of data in text form, a computing environment that supports more than one natural language is a must. This need is even more apparent in networked multiuser environments. A web search engine, as an access point to a global network of information, is much more useful if it can find documents written in different languages. As another example, the compilation of an online Korean-Chinese dictionary is best done in an environment that supports both languages. Even a personal phone directory application may need to handle names and addresses in a foreign language.

The issue of multilingual support in operating systems and applications has appeared and reappeared under many different headings: internationalization (also called i18n, since there are 18 characters between the initial "i" and the final "n" in the word), localization (l10n for the same reason), double byte character set (DBCS) support, locales, language toolkits, just to name a few. Whatever their names are, they are actually trying to tackle the same set of problems, though in different ways: to support computing environments that handle more than one natural language. Besides the ability of displaying characters correctly (output), a computing environment supporting a particular language should also provide methods for users to input its characters (input) and give a culture-specific interface using that language (user interface). A system that provides a multilingual environment supports more than one natural language at the same time. Although a specific set of language and cultural settings is usually used for the user interface in such a system, the input and output of text in more than one language should be supported in a truly multilingual system.

Support of a multilingual computing environment is by no means trivial. In Section 2, we discuss some of the problems that have to be solved. Current "solutions" to the problem can be put under four categories, for which we will introduce in Section 3. As we will discuss, they often only tackle part of the problem and are thus inadequate when applied alone. To make the computing environment truly multilingual, simultaneous use of more than one of these solutions is required. Yet, subtle interdependencies in their implementations often make it difficult to do so. For example, system administrators need to install and maintain multiple packages for langauage support, programmers need to use a lot of interdependent library routines to write a multilingual program, and users are often bothered by the technical questions asked by application programs.

We believe that a unifying framework of multilingual support is needed for a system to be truly user-, programmer- and system administrator-friendly. Our design of one such framework is outlined in Sections 4 to 7. We take a system-resource point of view for multilingual environment support: the choice of different language environments is seen as the use of different sets of resources in the system. A distributed system-level database, called MLDB, is thus designed to handle these resources, and a set of supporting routines, called MLLib, is used to provide programmers an application programming interface.

2 Practical issues

To support a multilingual environment, a number of practical issues need to be considered. Input, output and the support of coded character sets are among the most important ones. Here, we briefly discuss some of the more important issues.

Coded character sets such as ANSI X3.4 ("US-ASCII")[1], ISO 8859-1 ("Latin-1")[3], and Unicode [2] are used to represent characters in a computer. To interoperate with existing systems and to support more than one language, one coded character set is often not enough. Code structuring and extension techniques can be used to extend the number of characters a fixed code-width coded character set can hold. For example, the international standard ISO/IEC 2022 [4] specifies one such technique that makes use of escape codes to switch between code elements (blocks of character codes). Alternatively, multiple coded character sets, or a universal coded character set that contains all the scripts, can be used. Higher-level protocols that tag textual data with the coded character set it belongs to can also be used for multiple coded character set support. As a last resort, the system can require the user to run a program to convert their textual data from one coded character set to another.

Input methods: Input methods enable users to enter characters in the supported languages. For languages with a small number of alphabets, such as English and French, this is often easy. A key on the keyboard simply maps to a character in the alphabet. However, for Asian and Indic languages whose number of characters is large, more established methods for input are needed. Multiple keystrokes map onto one character in these languages, and user feedback is usually needed in the middle of entry. Different schemes for mapping the keystrokes to characters thus constitute different input methods. For example, 拼音 PinYin, 注音 ZuYin, 倉頡 CangJei and 四角 Four Corner are commonly used methods for Chinese input. PinYin and ZuYin use phonetic transcriptions of characters for input. The former uses English characters for the transcription ("romanization") and the latter uses a special set of phonetic characters. In contrast, structural, rather than phonetic, information is used in CangJei and Four Corner input methods. CangJei input method assigns almost all Chinese characters a unique key sequence by breaking them into components. Four Corner method gives each character a number which represents the structure of the four corners of a Chinese character.

Note that keyboard entry is not the only way characters can be input to the system. Handwritten input and speech recognition techniques can be considered as input methods in the general sense.

Text output includes both hardcopy generation and screen display. High-quality output is often required in hardcopies, but the time constraints for generating them are usually not tight. In contrast, real-time updates are often needed for display output. Hence, it is often done incrementally; only the part of the screen that has been

modified is redrawn. Moreover, since the quality of screen display is often low and its resolution fixed, further speed optimization techniques can be used. For example, fixed-size bitmaps of character glyphs (forms of shapes) can be generated in advance. Output of a character would then correspond to the copy of its bitmap to the screen.

Among the problems in the output of multilingual text, the handling of context dependency and directionality are perhaps the most difficult ones.

Directionality: The assumption that all scripts run from left to right in lines from top to bottom does not always hold in a multilingual environment. For example, Japanese and Chinese scripts can run from top to bottom in lines from right to left.

Context dependency: In some scripts such as Arabic and Sanskrit, the display order and the shape of a character depend on the characters around it in the logical order. A coded character no longer corresponds to a fixed glyph. Thus, simple schemes for output, such as the use of the character code as an index to a glyph, cannot be used.

We should note that for some coded character set standards context dependency and directionality considerations are explicitly mentioned in their conformance clauses. For example, character combination and bidirectional behavior are normative character properties and behaviors of the Unicode 2.0 standard [2, Chapter 3].

User interfaces using text have to display strings in the right language, which, in turn, depends on the user settings. Some of the issues on the design of a multilingual user interface include the assignment of shortcut keys, the set of allowable filenames, and the use of graphics elements. The reader is referred to literatures such as [5], [13], and [12] for a more detailed discussion on the practical issues involved.

Application programming interfaces (APIs) are needed for programmers to develop multilingual applications. Besides other reasons, the breakdown of the assumption that a character is 8 bit in size leads to the change or redesign of the API that handles characters and strings. Other APIs are also affected. Functions for input and output have to be modified to handle multilingual text. Network code that makes use of information in text form (e.g., domain name service) may also need to be changed. Indeed, all libraries that make use of character- and string-processing functions may need to be modified or recompiled. Since incompatible changes of APIs will require modification and recompilation of all programs that use them, it is difficult to define or extend an API for a multilingual environment that is compatible with existing systems.

Now we have described some of the problems we face in supporting a multilingual environment, let us review some of the current approaches to their solutions.

3 Models for natural language support

Mechanisms such as the ANSI-C locale model or the X/Open internationalization model [11] allow the user to specify language and cultural settings by using just one symbol. For example, the locale `en_US` indicates an environment that uses American (`US`) English (`en`). Indeed, these mechanisms allow the user to indirectly specify the behavior of language-dependent program entities (code and data). Existing approaches for implementing this language-dependent behavior often fall into one of the following four categories:

Conversion of data is often needed for systems that only support a limited number of coded character sets. Internal code conversion applications are used to convert text files to the coded character set accepted by the system. From the user's point of view, the conversion process may be explicit of implicit. Explicit conversion requires user intervention and is the most popular way of implementing "language enablers", a layer of software above the operating system for the support of a particular language. TwinBridge and RichWin are examples of Chinese language enablers over Microsoft Windows. Implicit conversion, in which no user intervention is needed, is possible if enough information about the system and the text is known. This information includes the coded character sets the system supports, and the coded character set the text data uses.

Since explicit code conversion requires users to know the technical details (e.g., the coded character set used) of the documents they are processing, it is not very user-friendly. However, it is unavoidable if the system does not have enough information. Indeed, utilities for explicit code conversion are often provided in systems that support implicit code conversion.

Conditional inclusion is the approach often used in the development of localized software, that is, software that is customized for a certain language environment. Instead of maintaining multiple sets of source code of the same program for different language environments, source code for a certain language environment is conditionally included at compile time by means of compiler directives.

With the use of the conditional inclusion technique, the resulting code is often smaller than that generated by the use of the selection technique described below. Also, only one set of source code needs to be maintained. However, the downsides are that the source would usually be less readable, the language environment is fixed at compile time, and differently localized software needs to be separately tested.

Selection: If the number of language environments supported is limited, program entities that depend on the language environment can be selected at run time by the use of conditional statements in the program. Effectively, one more dimension, namely the language environment, is associated with the program entities so that they can be referenced environment-dependently at run time. The use of the selection technique may or may not be transparent to the user. For example, the web browser `netscape`,

which allows the user to explicitly select the document encoding of a homepage, uses the selection technique which is not transparent to the user. Nontransparent selection, besides requiring user intervention, often requires the users to know technical information about the way the system supports multiple languages. Similar to the case of conversion, transparency can be obtained if we have enough information about the environment and the data we are handling.

Indirection: Program entities that should behave differently under different language environments are factored out, stored separately, and referenced indirectly in the program. The selection of program entities in the language environment dimension, that is, the set of factored-out program entities to use, is determined at startup or run time by the language environment. Message catalog implementations such as `catgets` [15] and `gettext` follow this principle. Program code can also be indirectly selected by the use of dynamic link libraries. The article [14] reports an implementation using such an approach.

Although most implementations of language-dependent software fall into one of the four approaches covered above, most of them focus on only part of the problem. Conversion mainly addresses the interoperability issue of passive data (character codes), Conditional Inclusion simplifies the maintenance of source code of the same program though the language environment is fixed at compile time, Selection helps the choice of program entities dynamically, and Indirection allows both code and data to be replaced without much increase in the size of executables.

Although more than one solution can be applied at the same time for multilingual support, in practice, subtle interdependencies between the solutions make the implementation more involved. For example, a system may support two coded character sets and allow users to select the one to use. If the system supports message catalog using the X/Open `catgets` mechanism, a programmer has to explicitly change the message catalog whenever the coded character set the system uses changes. Handling subtleties like this is not programmer-friendly. It would not only distract programmers from solving the problem at hand, but also discourage them from writing multilingual programs. Moreover, the system administrator may have to install and maintain multiple software packages for language support; it is not very system administrator-friendly either. A unifying framework with the goals of user-, programmer- and system administrator-friendliness is needed in a truly multilingual environment.

4 A framework for multilingual support

The goals in our design of the framework are user-, programmer- and system administrator-friendliness. These criteria are outlined in the following subsections.

4.1 User-friendliness

Users should not be made to answer technical questions regarding the data they are handling whenever possible. For example, they should not be made to specify the coded character set a document uses during editing or viewing. Also, a flexible system should allow its users to customize their language environments.

4.2 Programmer-friendliness

Programmers should concentrate on solving the problem given to them. They should not be forced to handle language support details such as the mode of user interaction and character encoding schemes. However, they should be given access to these details if they want to. In other words, multilingual text handling APIs should be nonintrusive, yet allow access to sufficient details. Otherwise, it would discourage programmers from writing software for a multilingual environment.

A two-line code for "ask the user to input a string" should not be expanded into a 200-line code when the word "multilingual" is added into it. As an example, the sample code in X11R5 Xlib Programming Manual that "creates a very simple window, connects to an input method and displays composed text" [9]— which essentially means "print out whatever is input" — is six pages long. A more established code for doing essentially the same thing in Programmer's Supplement for Release 6 [10] is even longer — 14 pages. These are rather programmer-hostile. Most of the code in these sample programs deals with details such as connection to input method servers, setting the interaction style, allocation of window areas for user feedback, event filtering and handling, and character lookup. These should be replaced by simple one- or two-line function calls unless the programmer wants to have full control over these details.

Using multilingual text I/O as an example, we believe that simple code like the one shown in Figure 1 is enough for a program to print out whatever is input in a multilingual environment. The class `MLString` shown there hides all the details of multilingual string handling. A multilingual string is handled like any other string type. Whether it holds characters in Unicode, ASCII or Latin-1 should be of no concern to the programmer who simply wants multilingual I/O. To provide programmers with a finer control over multilingual I/O, a set of support functions and manipulators can be implemented on `MLString` and the stream classes `cin` and `cout`. For example, one of the output manipulators can be used to enable or disable context-dependent rendering of text in the output stream. Yet, to be friendly to programmers, whether to use these support routines for a finer control of multilingual I/O should be left to their decisions.

4.3 System administrator-friendliness

The implementation of the framework should not require too much intervention from the system administrator. Use of multiple copies of software for different language environments is thus out of the question. For a certain site, centralized administration of elements

```
#include <iostream.h>
#include <MLString.h>
int main(void)
{
  MLString mls;    // Multilingual string type MLString
  cin >>mls;       // Input statement
  cout<<mls;       // Output statement
  return 0;
}
```

Figure 1: C++-style program source that prints out whatever is input

of multilingual support is preferred. This simplifies the job of system administration. It is less likely for the system administrator to forget to update a dependent component for multilingual support in an upgrade process, which in consequence breaks working software.

With these design goals in mind, we reckon that the existing approaches for multilingual support, as discussed in Section 3, should be unified under a single framework.

In our design of the framework, we take a system-resource point of view of a multilingual environment: the choice of different language environments is seen as the use of different sets of resources in the system. For example, the choice of an American English environment requires the support of a character set containing all the English alphabets such as US-ASCII, a corresponding font, a simple input method that maps a keystroke to a character, no directionality and minimal context-dependency support, and so on. In our design, the correspondence between the language environment and the resource set is stored in a specialized database, to be discussed next.

5 A Database for multilingual information

To specify a set of resources required for the support of a particular natural language, a system-level database specialized in the handling of multilingual environment information can be used. Specialized system-level databases are not uncommon in existing systems. Network Information Service (NIS, formerly called YP) and Network Information Service Plus (NIS+)[6], both designed by Sun Microsystems Inc., are examples of such databases.

Since the number of natural language computing environments can be arbitrary, it is impractical or impossible to provide the resources to support all of them in a single system. So, to provide the best support of a truly multilingual environment, there should be a way to obtain the needed resources if they are not available locally. A machine should be able to share its resources with its network peers. That way, the system would be much more system administrator-friendly because only those resources not found in the whole network need to be installed. A distributed database should be used so that a system can obtain

information about the resources its peers have. Again, distributed system databases are not rare in existing systems. Servers that provide Domain Name Service (DNS) [7][8] are actually distributed system databases.

Now we have established that a distributed system-level database specialized in the handling of information about language environments is needed for a truly multilingual system. Hereafter, we will call this database MLDB. Note that, besides passive data, active data such as code for subroutines, or their associated information, can also be stored in an MLDB. In our design, the information an MLDB can hold includes the followings:

Information about coded character sets: This includes the *name* and *size* of coded character sets, and the *number of bytes* a coded character would occupy. The *escape sequences* for code elements (block of character codes) in code structuring and extension standards such as ISO/IEC 2022 [4] are useful for implementors of libraries such as `MLString` shown above. *Code attributes* such as the name and the range of character blocks are also used for coded character set support. *Conversion tables* are essential for code conversion applications.

Localization information: This includes the *names* and *aliases* of locales, *format specifications* of date, time, numeric and monetary values under different language and cultural settings, and *collation order* of characters. They can be used to specify the language and cultural preferences in user interfaces.

Text handling code: Program code whose behavior depends on the language environment, such as that for character output, can be shared by clients of MLDBs. *Meta-information* about them, such as the names of the dynamic link libraries they reside on is also useful.

Text messages: Text messages and their *translations*, in a way similar to message catalogs, are contained in MLDB. Similar to GNU `gettext`, the current language environment, together with the untranslated text, can be used as the key to retrieve the translated messages.

Input methods and their associated data: Besides the code for input methods, data such as the associated *trie* or *state transition tables* can be shared by input methods. More general data about languages, such as the *character* or *word frequency table* for a certain language in a certain context, can also be made available by storing them into MLDB. Existing input methods can make use of this data to provide better user feedback. It also facilitates the design of new input methods.

Output routines: Similar to input methods, output routines are often language environment-dependent. Rendering rules about the handling of context dependent and directional text can be shared by clients who need multilingual output.

Addresses: Existing servers such as *font servers*, *input method servers*, or *rendering engines* already provide a lot of services. Their addresses can be stored in MLDB so that they can be looked up easily by clients.

An MLDB that provides information about multilingual environments can be used by application programs. A set of API is thus needed for the application programmers to interface with MLDB.

6 Multilingual support libraries

To use the services for multilingual support, an application program has to contact an MLDB for information. A set of software libraries that provides an application programming interface (API) is thus needed. Functions in low-level multilingual support libraries (hereafter called low-level MLLib) provide APIs for applications to communicate directly with MLDBs in the network using a predefined protocol. They provide application programmers a structured way to retrieve items stored in the MLDBs using a simple query-response model. With these APIs, queries similar to the followings can be answered:

- locale="fr",messagekey="Shutdown",translation=?

  Find the translation of the message "Shutdown" in the French locale.

- charset_alias="US-ASCII",charset_name=?

  Find the name of the coded character set whose alias is "US-ASCII".

Besides acting as a liaison between the application and the MLDB, low-level MLLib is also used to provide other facilities. For example, results obtained from an MLDB can be cached so that the response time to future identical queries is shortened. Also, when requested to provide certain locally unavailable resources, low-level MLLib can transparently obtain the missing information from its networked peers. Support of distributed resource procurement thus becomes a facility of the low-level MLLib.

Although low-level MLLib simplifies the communication between application programs and the MLDBs, it would not be very programmer-friendly if it is the only means of handling language-related operations. Programmers should not be made to care about the communication between their application and the MLDBs. Higher level support routines are needed. These are provided by high-level MLLib, a set of library routines that make use of the information obtained by low-level MLLib calls to do the jobs in a specialized way. The `MLString` class shown in Figure 1 is an example of code in such a high level library. It makes use of low-level MLLib functions to obtain information such as the coded character sets supported and the input methods available so that it can handle multilingual I/O in its own way. Other multilingual text handling functions can also be implemented this way. For example, a function that provides explicit code conversion facilities can be implemented as follows:

```
MLString convert(MLString st, Codeset a, Codeset b)
// Converts the string st from coded character set a to coded character set b
Query MLDB the source and destination codesets of the
    available code conversion tables
```

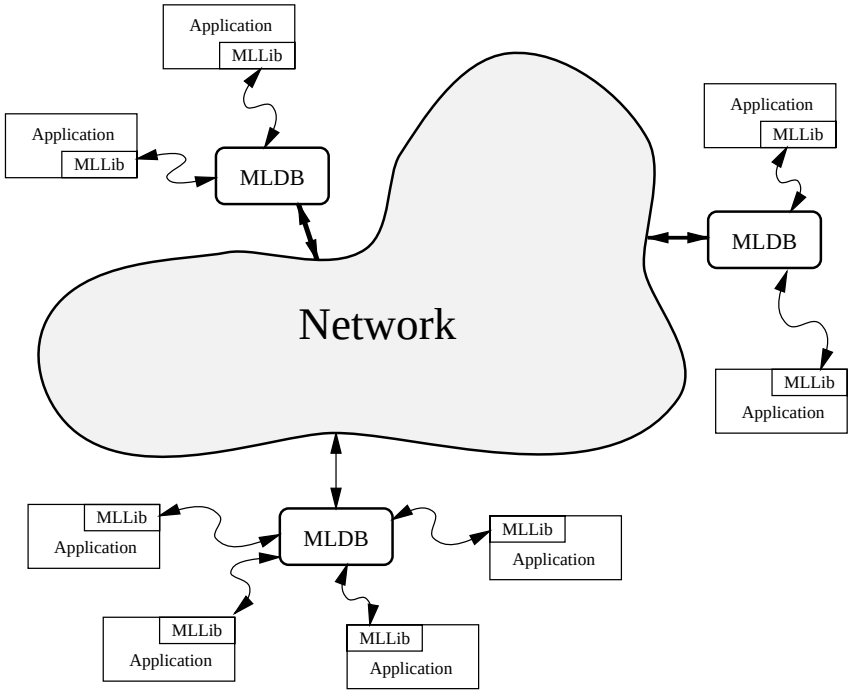

Figure 2: An architecture of a specialized distributed multilingual system database

Find the least cost conversion path
(e.g., the shortest path with least information lost,
codeset conversion from a to c to b
is preferred to that from a to d to e to b)
Retrieve the required code conversion tables
Do the conversion using the tables

Besides being programmer-friendly, the combined use of MLDB and MLLib would also ease the job of system administrators because MLDB handles the interdependencies between natural language support modules. The distributed database architecture, which allows application programs to obtain resources from the network, makes locally unavailable resources available. That way, a system needs not have the whole set of resources for the support of all possible languages. This makes system administration much easier.

7 The architecture

The general architecture for the support of multilingual computing environment is shown in Figure 2. High- and low-level MLLib's are bunched together and is called MLLib in the figure. Application codes linked with MLLib can make use of the information about the language settings from the runtime environment to query MLDB and obtain the needed resources. In the process of handling multilingual text, MLLib transparently obtains the

needed resources, such as font glyphs and input methods, on behalf of the application using them. Intermediate transformation of multilingual data, such as code conversion, can also be done transparently if possible. Technical details on multilingual support are thus hidden from the application users.

With this architecture, multilingual applications can be developed relatively easily by using MLLib. Users are hidden from the details of how the system supports multilingual text, and the system administrator's job is eased because the maintenance of multiple natural language environments is not needed.

We can compare our architecture of multilingual support to the architecture for graphical user interface in X Window System. Low-level MLLib provides an API for application programmers to obtain information from MLDB, while Xlib[9] provides an API for communicating with the X server. High-level MLLib simplifies programming by using information obtained from MLDB to implement multilingual support functions, while X Toolkit simplifies programming by using facilities provided by X servers to implement widgets such as scroll bars. The difference is seen when we compare MLDB with the X server. The former manages data and meta-information required for language support while the latter mainly manages the display.

8 Summary

The issue of supporting a multilingual computing environment, such as the support of coded character sets, input, output, user interface and API, has been covered in this paper. For multilingual applications, program entities, that is, code and data, are often different under different language environments. Existing approaches for implementing these language-dependent behavior often fall into one of the four categories introduced in this paper: Conversion, Conditional Inclusion, Selection, and Indirection. Yet, these approaches focus on only part of the problem. Conversion mainly addresses the interoperability problem of passive data, Conditional Inclusion simplifies the maintenance of source code of the same program though the language environment is fixed at compile time, Selection helps the choice of program entities dynamically, and Indirection allows both code and data to be replaced without much increase in the size of executables. A user-, programmer- and system administrator-friendly framework that unifies these approaches is thus needed.

As we argued, the choice of a language environment can be seen as the use of a particular set of resources in a system. Thus, our approach to the design of a unified framework is to, first of all, develop a distributed system database, MLDB, that is specialized in handling these resources. The second step is to interface MLDB to application programs via a set of software libraries, MLLib. Besides acting as a liaison between application programs and MLDBs, MLLib also hides the details of multilingual environment handling from application programmers. It also provides support functions that allow the application programmers to write codes that do multilingual I/O easily.

With this architecture, multilingual applications can be developed relatively easily by using MLLib. Users are hidden from the details of how the system supports multi-

lingual text. Moreover, maintenance of multiple natural language environments is not needed. Our approach thus achieves the goals of user-, application programmer-, and system administrator- friendliness.

References

[1] American National Standards Institute. *ANSI X3.4-1977: American National Standard Code for Information Interchange*, 9 June 1977.

[2] The Unicode Consortium. *The Unicode Standard, Version 2.0*. The Unicode Consortium, July 1996. ISBN 0-201-48345-9.

[3] International Organization for Standardization. *ISO 8859-1:1987: Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, 15 February 1987.

[4] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 2022:1994: Information technology – Character code structure and extension techniques*, 1994.

[5] Nadine Kano. *Developing international software for Windows 95 and Windows NT*. Microsoft Press, 1995. ISBN 1-55615-840-8.

[6] Chuck McManis and Sagib Jang. Network information services plus: A white paper. Technical report, Sun Microsystems Inc., 1991.

[7] P. Mockapetris. Domain names - concepts and facilities. Request for Comments (Standard) RFC 1034, Internet Engineering Task Force, November 1987. Obsoletes RFC0973; Updated by RFC1101.

[8] P. Mockapetris. Domain names - implementation and specification. Request for Comments (Standard) RFC 1035, Internet Engineering Task Force, November 1987. Obsoletes RFC0973; Updated by RFC1348.

[9] Adrian Nye. *Xlib Programming Manual*, volume 1 of *The Definitive Guides to the X Window System*. O'Reilly & Associates, Inc., third edition, 1992.

[10] Adrian Nye. *Programmer's Supplement for Release 6 of the X Window System*. The Definitive Guides to the X Window System. O'Reilly & Associates, Inc., first edition, September 1995.

[11] Wendy Rannenberg and Jürgen Bettels. The X/Open internationalization model. *Digital Technical Journal*, 5(3):32–42, Summer 1993.

[12] Bill Tuthill. *Solaris International Developer's Guide*. SunSoft, 1993.

[13] Emmanuel Uren, Robert Howard, and Tiziana Perinotti. *Software Internationalization and Localization: an Introduction.* Van Nostrand Reinhold, 1993.

[14] Gayn B. Winters. International distributed systems — architectural and practical issues. *Digital Technical Journal*, 5(3):53–62, Summer 1993.

[15] X/Open group members. *System V Specification Supplementary Definitions*, volume 3 of *X/Open Portability Guide.* Elsevier Science Publishers B.V., January 1987. ISBN 0-444-70176-1.