

Source: <http://research.compaq.com/wrl/DECarchives/DTJ/DTJB05/DTJB05SC.TXT>
File Creation Date: July 28, 2002

INTERNATIONAL DISTRIBUTED SYSTEMS -- ARCHITECTURAL AND PRACTICAL ISSUES

By Gayn B. Winters

ABSTRACT

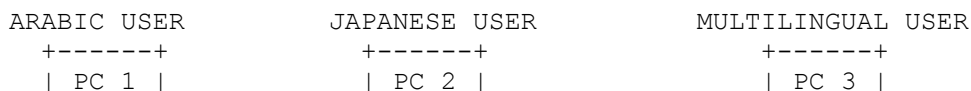
Building distributed systems for international usage requires addressing many architectural and practical issues. Key to the efficient construction of such systems, modularity in systems and in run-time libraries allows greater reuse of components and thus permits incremental improvements to multilingual systems. Using safe software practices, such as banishing the use of literals and parameterizing user preferences, can help minimize the costs associated with localization, reengineering, maintenance, and design.

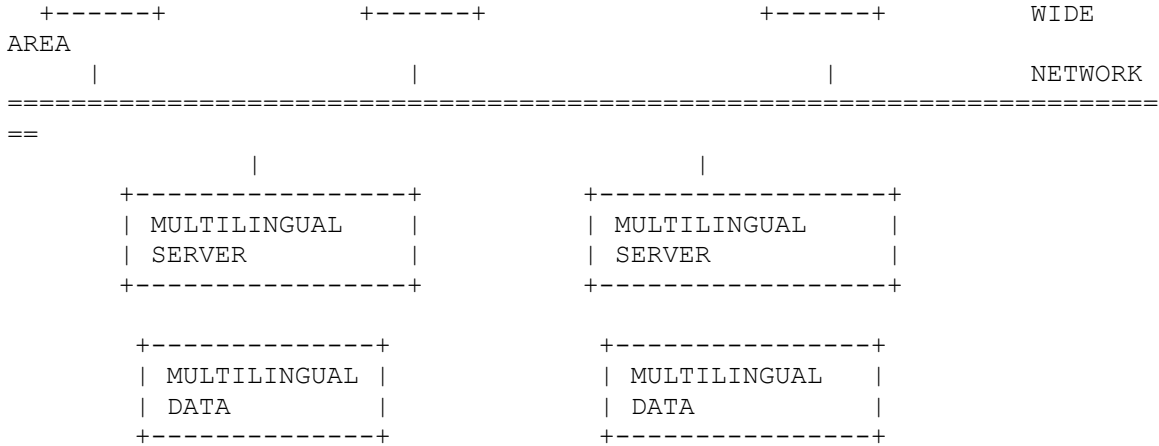
INTRODUCTION

The worldwide deployment of computer systems has generated the need to support multiple languages, scripts, and character sets simultaneously. A system should focus on natural ease of use and thus allow end users to read system messages in the language of their choice, to have natural menus, forms, prompts, etc., and to enter and display data in their preferred presentation form.

Digital envisions a computer system that not only is distributed but is distributed geographically across the world. A single site may have end users with varying language and cultural preferences. For example, a Japanese bank in Tel Aviv may have employees whose native languages are Arabic, English, Hebrew, Japanese, or Russian, and may conduct business in one or several of these languages. Figure 1 could represent a portion of their network. The client software, e.g., a mail client and the local windowing system, could be completely monolingual. Networking, database, and printing services, for instance, should be multilingual in that they support the various end users by providing services independent of the natural languages, scripts, or character sets used.

Figure 1 A Portion of a Multilingual Network





This paper surveys many of the architectural and practical issues involved in the efficient construction of international distributed systems. We begin by discussing some economic issues and pitfalls related to localization and reengineering. Many of these topics can be addressed by straightforward good engineering practices, and we explore several important techniques. The structure of application-specific and system-level run-time libraries (RTLs) is a key issue. We therefore devote several sections of this paper to preferred RTL structures, data representations, and key RTL services. Distributed systems cause some special problems, which we briefly discuss, commenting on naming, security, management, and configuration. In particular, a desire for client software designed for monolingual distributed systems to work without change in a multilingual distributed system led to a new system model. In the model, the host servers and the system management provide the interfaces and conversions necessary for these clients to interface with the multilingual world. Finally, we observe that all the preceding techniques can be delivered incrementally with respect to both increasing functionality and lowering engineering cost.

LOCALIZATION AND REENGINEERING

When a system component is productized for some local market, the process of making it competitive and totally acceptable to that market is called localization. During this process, changes in the design and structure of the product may be required. These changes are called reengineering. For example, U.S. automobiles whose steering linkages, engine placement, console, etc., were not designed to allow the choice of left- or right-hand steering were not competitive in Japan. Reengineering these automobiles for right-hand steering was prohibitively expensive, so manufacturers had to redesign later models.

Computer systems have problems similar to the automobile left-hand-right-hand steering problem. A good architecture and design is necessary to avoid expensive reengineering during localization. The following are examples of areas in which a localization effort may encounter problems: user-defined

characters and ligatures; geometry preferences, such as vertical or right-to-left writing direction, screen layout, and page size; and policy differences, such as meeting protocols and required paper trails. Building limiting assumptions into a software or hardware product can often lead to costly reengineering efforts and regional time-to-market delays.

On the other hand, an internal survey of reengineering problems associated with Digital's software indicates that simple, easy-to-avoid problems are strikingly frequent. In fact, it is amazing how many ways a U.S. engineer could find to make use of the (ultimately erroneous) assumption that one character fits into one 8-bit (or even more constrictive, one 7-bit) byte!

SAFE SOFTWARE PRACTICES

Many well-known, straightforward programming practices, if adopted, can dramatically reduce reengineering efforts.[1-7] Even for existing systems, the cost of incrementally rewriting software to incorporate some of these practices is often more than recovered in lower maintenance and reengineering costs. This section discusses a few key practices.

Probably the most fundamental and elementary safe software practice is to banish literals, i.e., strings, characters, and numbers, from the code. Applying this practice does not simply redefine YES to be "yes" or THREE to be the integer 3. Rather, this practice yields meaningful names, such as `affirmative_response` and `maximum_alternatives`, to help anyone who is trying to understand how the code functions. Thus, not only does the practice make the code more maintainable, but it also makes it easier to parameterize or generalize the data representation, the user interface preferences, and the functionality in ways the original programmer may have missed. These definitions can be gathered into separate declaration files, message catalogs, resource files, or other databases to provide flexibility in supporting clients of different languages.

The abstraction of literals extends to many data types. In general, it is best to use opaque data types to encapsulate objects such as typed numbers (e.g., money and weight), strings, date and time of day, graphics, image, audio, video, and handwriting. Providing methods or subroutines for data type manipulation conceals from the application how these data types are manipulated. The use of polymorphism can serve to overload common method and operation names like `create`, `print`, and `delete`. Support for multiple presentation forms for each data type should allow additional ones to be added easily. These presentation forms are typically strings or images that are formatted according to end-user preferences. Both input and output should be factored first into transformations between the data type and the presentation form, and then into input and output on the presentation form. For example, to input a date involves inputting and parsing a string that represents a presentation form of the date, e.g., "17 janvier 1977," and computing a value whose data type is `Date`.

The concepts of character and of how a character is encoded inside a computer vary dramatically worldwide.[2,7-11] In addition, a process that works with a single character in one language may need to work with multiple characters in another language. One simple rule can prevent the problems that this variation can cause: Banish the Character data type from applications, and use an opaque string data type instead. This rule eliminates the tempting practice of making pervasive use of how a character is stored and used in the programmer's native system. The Array of Character data type is nearly as insidious, because it is tempting to use the i th element for something that will not make sense in another natural language. One should only extract substrings $s[i:j]$ from a string s . Thus, when in a given language the object being extracted is only one code point $s[i:i]$, the extraction is obviously a special case. The section Text Elements and Text Operations discusses this concept further.

Another safe software practice is to parameterize preferences, or better yet, to attach them to the data objects. As discussed previously, a "hardwired" preference such as writing direction invariably becomes a reengineering problem. The language represented by the string, the encoding type, the presentation form of the object, and the input method for the object are all preferences. In servers and in all kinds of databases, tagging the data with its encoding type is desirable. In general, the data type of the object should contain the preference attributes. The client that processes the object can override the preferences.

Geometry preferences should be user selectable. Some geometry preferences affect the user's working environment, e.g., the ways in which dialog boxes work, windows and pop-up menus cascade, and elevator bars work.[1] These preferences are almost always determined by the end user's working language. Other geometry preferences relate to the data on which the user is working, e.g., paper size, vertical versus horizontal writing (for some Asian languages), how pages are oriented in a book, layouts for tables of contents, and labels on graphs.

Computer programs, in particular groupware applications, mix policy with processing. "Policy" refers to the sequence or order of processing activities. For example, in a meeting scheduler, can anyone call a meeting or must the manager be notified first? Is an invoice a request for payment or is it the administrative equivalent of delivered goods requiring another document to instigate payment? Often such policy issues are not logically forced by the computation, but they need to be enforced in certain business cultures. A sequence of processing activities that is "hardwired" into the program can be very difficult to reengineer. Thus, policy descriptions should be placed into an external script or database. The advent of workflow controllers, such as those in Digital's EARS, ECHO, and TeamRoute products, makes it easy to do this.

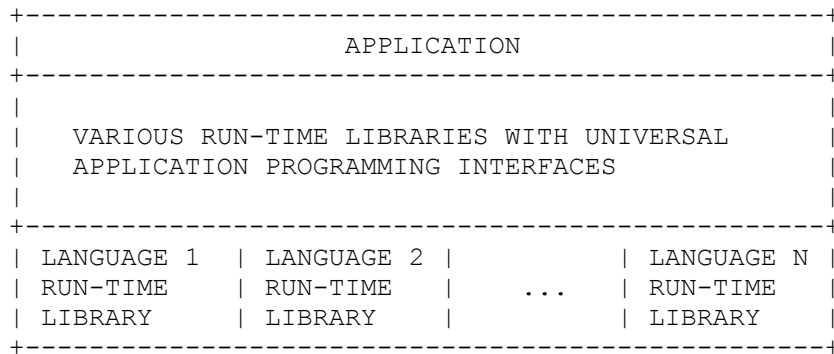
Applications should not put date formatting, sorting, display, or input routines into their mainline code. Often such operations

have been coded previously, and a new application's code will probably not be international and may well contain other bugs. Therefore, programmers should construct applications to use, or more precisely reuse, run-time libraries, thus investing in the quality and the multilingual and multicultural capabilities of these RTLs. When the underlying system is not rich enough and/or competition dictates, the existing RTL structures must be augmented.

RUN-TIME LIBRARY STRUCTURE

A common theme for internationalizing software and for the safe programming practices discussed in the previous section is to keep the main application code independent of all natural language, script, or character set dependencies. In particular, the code must use only RTLs with universal application programming interfaces (APIs), i.e., the name of the routine and its formal parameter list must accommodate all such variants. Digital's early localization efforts typically made the mistake of replacing the U.S.-only code with code that called RTLs specific to the local market. This practice generated multiple versions of the same product, each of which needed to be changed whenever the pertinent part of the U.S. version was changed. A better structure for run-time libraries is shown in Figure 2.

Figure 2 Modular Run-time Library Structure



The application illustrated in Figure 2 calls an RTL routine through the routine's universal APIs. This routine may in turn call another language-specific routine or method, or it may be table driven. For example, a sort routine may be implemented using sort keys rather than compare functions for better performance. With this structure, localization to a new language involves only the addition of the new language-specific RTL or the corresponding new table entries.

Note that the application must pass sufficient structure to the RTL to guarantee that the APIs are universal. For example, to sort a list of strings, a call

```
sort_algorithm(list_pointer, sort_name, sort_order)
```

could be created. The `sort_order` parameter is of the type `{ascending,descending}`. The `sort_name` parameter is necessary because in many cultures numerous methods of sorting are standard.[1,12] In some RTL designs, notably those specified by X/Open Company Ltd., these extra parameters are passed as global variables.[5,6,7] This technique has the advantage of simplifying the APIs and making them almost identical to the APIs for the U.S. code. Such RTLs, however, do not tend to be thread-safe and have other problems in a distributed environment.[5,13,14] An alternative and far more flexible mechanism is more object oriented -- using a subtype of the List of String data type when alternate sorts are meaningful. This subtype has the additional information (e.g., `sort_name` and `sort_order`) used by its `Sort` method.[12,14]

The next three sections discuss the organization and extensibility of RTLs with this structure.

DATA REPRESENTATION

Data representation in RTLs incorporates text elements and text operations, user-defined text elements, and document interchange formats.

Text Elements and Text Operations

A text element is a component of a written script that is a unit of processing for some text operation, such as sorting, rendering, and substring search. Sequences of characters, digraphs, conjunct consonants, ligatures, syllables, words, phrases, and sentences are examples of common text elements.[10,15] An encoded character set *E* represents some particular set of text elements as integers (code points). Typically, the range of *E* is extended so that code points can represent not only text elements in multiple scripts but also abstractions that may or may not be part of a script, such as printing control codes and asynchronous communication codes.[16] More complex text elements can be represented as sequences of code points. For example, \hat{U} may be represented by two code points `<U> <^>`, and a ligature such as {O joined with E} may be represented as three code points `<O> <joiner> <E>`, where a "joiner" is a special code point reserved for creating text elements. Less complex text elements, i.e., subcomponents of the encoded text elements, are found by using the code point and the operation name to index into some database that contains this information. For example, if `<é>` is a single code point for `é`, then the base character `e` is found by applying some function or table lookup to the code point `<é>`. The same is true for finding a code point for the acute accent. When a sequence of code points represents a text element, the precise term "encoded text element" is often abbreviated as "text element."

An encoded character set of particular importance is Unicode, which addresses the encoding of most of the world's scripts using

integers from 0 to $2^{16} - 1$. [11, 17] The Unicode universal character set is the basis of ISO 10646, which will extend the code point interval to $2^{31} - 1$ (without using the high-order bit). [9] Unicode has a rich set of joiner code points, and it formalizes the construction of many types of text elements as sequences of code points.

Processing text elements that are represented as sequences of code points usually requires a three-step process: (1) the original text is parsed into operation-specific text elements, (2) these text elements are assigned values of some type, and (3) the operation is performed on the resulting sequence of values. Note that each step depends on the text operation. In particular, a run-time library must have a wide variety of parsing capabilities. The following discussion of rendering, sorting, and substring searching operations demonstrates this need.

In rendering, the text must be parsed into text elements that correspond to glyphs in some font database. The values assigned to these text elements are indexes into this database. The rendering operation itself gets additional data from a font server as it renders the text onto a logical page.

The sorting operation is more complicated because it involves a list of strings and multiple steps. A step in most sorting algorithms involves the assignment of collation values (typically integers) to various text elements in each string. The parsing step has to take into account not only that multiple code points may represent one character but also that some languages (Spanish, for example) treat multiple characters as one, for the purposes of sorting. Thus, a sorting step parses each string into text elements appropriate for the sort, assigns collation values to these elements, and then sorts the resulting sequences of values. Note that the parsing step that takes place in a sorting operation is somewhat different from the one that occurs in a rendering operation, because the sort parse must sometimes group into one text element several characters, each of which has a separate glyph.

Searching a string *s* for a substring that matches a given string *s'* involves different degrees of complexity depending on the definition of the term "matches." The trivial case is when "matches" means that the substring of *s* equals *s'* as an encoded substring. In this case, the parse only returns code points, and the values assigned are the code point values. When the definition of "matches" is weaker than equality, the situation is more complicated. For example, when "matches" is "equal after uppercasing," then the parsing step is the same one as for uppercasing and the values are the code points of the uppercased strings. (Note that uppercasing has two subtle points. The code point for a German sharp *s*, $\langle\beta\rangle$, actually becomes two code points $\langle S\rangle\langle S\rangle$. Thus, sometimes the values assigned to the text elements resulting from the parse consist of more code points than in the original string. In addition, this substring match involves regional preferences, for example, uppercasing a French *é* is *E* in France and *É* in Canada.) The situation is similar when "matches" equals "equal after removing all accents or similar rendering

marks." A more complex case would be when s' is a word and finding a match in s means finding a word in s with the same root as s'. In this case, the operation must first parse s into words and then do a table or dictionary lookup for the values, i.e., the roots.

User-defined Text Elements

When the user of a system wishes to represent and manipulate a text element that is not currently represented or manipulated by the system, a mechanism is required to enable the user to extend the system's capabilities. Examples of the need for such a mechanism abound. Chinese ideograms created as new given names and as new chemical compounds, Japanese gaiji, corporate logos, and new dingbats are often not represented or manipulated by standard systems.

User-defined text elements cause two separate problems. The first problem occurs when E, the encoded character set in use, needs to be extended so that a sequence of E's code points defines the desired user-defined text element. The issues related to this problem are ones of registration to prevent one user's extensions from conflicting with another user's extensions and to allow data interchange.

The second, more difficult problem concerns the extensions of the text operations required to manipulate the new text element. For each such text operation, the parsing, value mapping, and operational steps discussed earlier must be extended to operate on strings that involve the additional code points of E. When tables or databases define these steps, the extensions are tedious but often straightforward. Careful design of the steps can greatly simplify their extensions. In some cases, new algorithms are required for the extension. To the extent that these tables, databases, or algorithms are shared, the extensions must be registered and shared across the system.

Document Interchange Formats

Compound documents (i.e., documents that contain data types other than text) use encoded character sets to encode simple text. Although many new document interchange formats (DIFs) will probably use Unicode exclusively (as does Go Computer Corporation's internal format for text), existing formats should treat Unicode as merely another encoded character set with each character set being tagged.[18] This allows links to be made to existing documents in a natural way.

Many so-called revisable DIFs, such as Standard Generated Mark-up Language (SGML), Digital Document Interchange Format (DDIF), Office Document Architecture (ODA), Microsoft Rich Text Format (RTF), and Lotus spreadsheet format (WKS), and page description languages (PDLs), such as PostScript, Sixels, or X.11, can be extended to provide this Unicode support by enhancing the attribute structure and extending the text import map

Strings(E)-->DIF for each encoded character set E. In doing so, however, many of the richer constructs in Unicode, e.g., writing direction, and many printing control codes are often best replaced with the DIF's constructs used for these features instead.[19] In this way, both processing operations are easier to extend and facilitate the layout functions DIF-->PDL and the rendering functions PDL-->Image.

PRESENTATION SERVICES

The practice of factoring input and output of data types into a transformation T<-->T_Presentation_Form and performing the I/O on the presentation form allows one to focus on each step separately. This factorization also clarifies the applicability of various user preferences, e.g., a date form preference applies to the transformation, and a font preference applies to how the string is displayed. As mentioned in the section Safe Software Practices, preferences such as presentation form are best attached to the end user's copy of the data. Data types such as encoded image, encoded audio, and encoded video pose few international problems except for the exchangeability of the encodings and the viability of some algorithms for recognizing speech and handwriting. Algorithms for presentation services can be distributed, but we view them as typically residing on the client.[20] In Figure 1, we presume that the local language PCs have this capability.

Input

Existing technology offers several basic input services, which are presented in the following partial list of device-data type functions:

- o Keystrokes-->Encoded Character
- o Image-->Encoded Image
- o Audio Signal-->Encoded Audio
- o Video Signal-->Encoded Video
- o Handwriting-->Encoded Handwriting

The methods for each input service depend on both the device and the digital encoding and often use multiple algorithms. Whereas for some languages the mapping of one or more keystrokes into an encoded character (e.g., [compose] + [e] + [/] yielding é) may be considered mundane, input methods for characters in many Asian languages are complex, fascinating, and the topic of continuing research. The introduction of user-defined text elements, which is more common among the Asian cultures, requires these input methods to be easily extendable to accommodate user-defined characters.

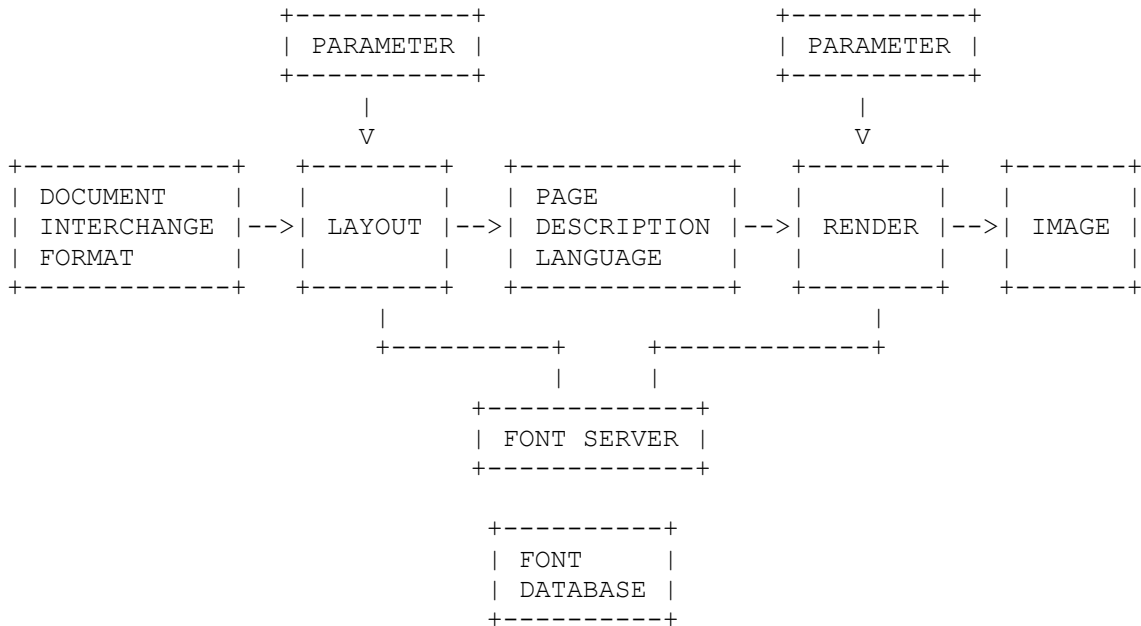
Output

The basic output services are similar to the input services listed in the previous section.

- o Strings-->Image
- o DIF-->PDL-->Image
- o Encoded Image-->Image
- o Encoded Audio-->Audio Signal
- o Encoded Video-->Video Signal
- o Encoded Handwriting-->Image

These output services also vary with encoding, device, and algorithm. Figure 3 illustrates the sequence DIF-->PDL-->Image. Optional parameters are permitted at each step. A viable implementation of Strings-->Image is to factor this function by means of the function Strings-->DIF, which is discussed in the Data Representation section. Alternatively, the data type Strings can be simply viewed as another DIF to be supported.

Figure 3 Layout and Rendering Services



A revisable document begins in some DIF such as plain text, Strings(Unicode), SGML, or DDIF. A layout process consumes the document and some logical page parameters and creates an intermediate form of the document in some PDL such as PostScript, Sixels, or even a sequence of X.11 packets. To accomplish this,

the layout process needs to get font metrics from the font server (to compute relative glyph position, word and line breaks, etc.). In turn, the rendering process consumes the PDL and some physical media parameters to create the image that the end user actually sees. The rendering process may need to go back to the font server to get the actual glyphs for the image. Rendering, layout, and font services are multilingual services. The servers for these services are the multilingual servers envisioned in Figure 1.

COMPUTATION SERVICES

To build systems that process multilingual data, such as the one shown in Figure 1, a rich variety of text operations is necessary. This section categorizes such operations, but a complete specification of their interfaces would consume too much space in this paper. Text operations require parsing, value mapping, and operational functions, as described earlier.

Text Manipulation Services

Text manipulation services, such as those specified in C programming language standard ISO/IEC 9899:1990, System V Release 4 Multi-National Language Supplement (MNLS), or XPG4 run-time libraries (including character and text element classification functions, string and substring operations, and compression and encryption services) need to be extended to multilingual strings such as Strings(Unicode) and other DIFs, and to various text object class libraries.[6,8,13]

Data Type Transformations

Data type transformations (e.g., speech to text, image-to-text optical character recognition [OCR], and handwriting to text) are operations where the data is transformed from a representation of one abstract data type to a representation of another abstract data type. The presentation form transformations $T \leftrightarrow T_Presentation_Form$ and the fundamental input and output services are data type transformations. Care needs to be taken when parameterizing these operations with user preferences to keep the transformation thread-safe. Again, this is best accomplished by keeping the presentation form preferences attached to the data.

Encoding Conversions

Encoding conversions (between encoded character sets, DIFs, etc.) are operations where only the representation of a single data type changes. For example, to support Unicode, a system must have for each other encoded character set a function $to_uni:Strings(E) \rightarrow Strings(Unicode)$, which converts the code points in E to code points in Unicode.[11] The conversion function to_uni has a partial inverse

from_uni:Strings(Unicode)-->Strings(E), which is only defined on those encoded text elements in Unicode that can be expressed as encoded text elements in E. If s is in Strings(E), then from_uni(to_uni(s)) is equal to s. Other encoding conversions Strings(E)-->Strings(E') can be defined as a to_uni operation followed by a from_uni operation, for E and E' respectively. Another class of encoding conversions arises when the character set encoding remains fixed, but the conversion of a document in one DIF to a document in another DIF is required. A third class originates when Unicode or ISO 10646 strings sent over asynchronous communication channels must be converted to a Universal Transmission Format (UTF), thus requiring Strings(Unicode)<-->UTF encoding conversions.

Collation or Sorting Services

Another group of computation services, collation or sorting services, sorts lists of strings according to application-specific requirements. These services were discussed earlier in the paper.

Linguistic Services

Linguistic services such as spell checking, grammar checking, word and line breaking, content-based retrieval, translation (when existent), and style checking need standard APIs. Although the implementation of these linguistic services is natural language-specific, most can be implemented with the structure shown in Figure 2.

Also, large character sets such as Unicode and other multilingual structures require a uniform exception-handling and fallback mechanism because of the large number of unassigned code points. For example, a system should be able to uniformly handle exceptions such as "glyph not found for text element." Mechanisms such as global variables for error codes inhibit concurrent programming and therefore should be discouraged. Returning an error code as the return value of the procedure call is preferred, and when supported, raising and handling exceptions is even better.

SYSTEM NAMING, SYNONYMS, AND SECURITY

The multilingual aspect of Unicode can simplify system naming of objects and their attributes, e.g., in name services and repositories. Using encoded strings tagged with their encoding type for names is too rigid, because of the high degree of overlap in the various encoded character sets. For example, the string "ABC" should represent one name, independent of the character set in which the string is encoded. Two tagged strings represent the same name in the system if they have the same canonical form in Unicode according to the following definitions.

Unicode has the property that two different Unicode strings, u

and *v*, may well represent the same sequence of glyphs when rendered.[11] To deal with this, a system can define an internal canonical form *c(u)* for a Unicode string *u*. *c(u)* would expand every combined character in *u* to its base characters followed by their assorted marking characters in some prescribed order. The recommended order is the Unicode "priority value." [11,21] The canonical form should have the following property: When *c(u)* is equal to *c(v)*, the plain text representations of *u* and *v* are the same. Ideally, the converse should hold as well.

Thus, *u* and *v* represent the same name in the system if *c(u)* is equal to *c(v)*. In any directory listing, an end user of a language sees only one name per object, independent of the language of the owner who named the object. Further restrictions on the strings used for names are desirable, e.g., the absence of special characters and trailing blanks. In a multivendor environment, both the canonical form and the name restrictions should be standardized. The X.500 working groups currently studying this problem plan to achieve comparable standardization.

Since well-chosen names convey useful information, and since such names are entered and displayed in the end user's writing system of choice, it is often desirable for the system to store various translations or "synonyms" for a name. Synonyms, for whatever purpose, should have attributes such as *long_name*, *short_name*, *language*, etc., so that directory functions can provide easy-to-use interfaces. Access to objects or attribute values through synonyms should be as efficient as access by means of the primary name.

In a global network, public key authentication using a replicated name service is recommended.[22] One principal can look up another in the name service by initially using a (possibly meaningless) name for the object in some common character set, e.g., {A-Z,0-9}. Subsequently, the principals can define their own synonyms in their respective languages. Attributes for the principals, such as network addresses and public encryption keys, can then be accessed through any synonym.

SYSTEM MANAGEMENT AND CONFIGURATION

The system management of a multilingual distributed system is somewhat more complicated than for a monolingual system. The following is a partial list of the services that must be provided:

- o Services for various monolingual subsystems
- o Registration services for user preferences, locales, user-defined text elements, formats, etc.
- o Both multilingual and multiple monolingual run-time libraries, simultaneously (see Figure 2)
- o Multilingual database servers, font servers, logging and queuing mechanisms, and directory services

- o Multilingual synonym services
- o Multilingual diagnostic services

Since a system cannot provide all the services for every possible situation, registering the end users' needs and the system's capabilities in a global name service is essential. The name service must be configured so that a multilingual server can identify the language preferences of the clients that request services. This configuration allows the servers to tag or convert data from the client without the monolingual client's active participation. Therefore, the name service database must be updated with the necessary preference data at client installation time.

Typically, system managers for different parts of the system are monolingual end users (see Figure 1) who need to do their job from a standard PC. Thus, both the normal and the diagnostic management interfaces to the system must behave as multilingual servers, sending error codes back to the PC to be interpreted in the local language. Although the quality of the translation of an error message is not an architectural issue, translations at the system management level are generally poor, and the system design should account for this. Systems developers should consider giving both an English and a local-language error message as well as giving easy-to-use pointers into local-language reference manuals.

Data errors will occur more frequently because of the mixtures of character sets in the system, and attention to the identification of the location and error type is important. Logging to capture offending text and the operations that generated it is desirable.

INCREMENTAL INTERNATIONALIZATION

Multilingual systems and international components can be built incrementally. Probably the most powerful approach is to provide the services to support multiple monolingual subsystems. Even new operating systems, such as the Windows NT system, that use Unicode internally need mechanisms for such support.[23] Multidimensional improvements in a system's ability to support an increasing number of variations are possible. Some such improvements are making more servers multilingual, supporting more multilingual data and end-user preferences, supporting more sophisticated text elements (the first release of the Windows NT operating system will not support Unicode's joiners), as well as adding more character set support, locales, and user-defined text elements. The key point is that, like safe programming practices, multilingual support in a distributed system is not an "all-or-nothing" endeavor.

SUMMARY

Customer demand for multilingual distributed systems is increasing. Suppliers must provide systems without incurring the costs of expensive reengineering. This paper gives an overview of the architectural issues and programming practices associated with implementing these systems. Modularity both in systems and in run-time libraries allows greater reuse of components and incremental improvements with regard to internationalization. Using the suggested safe software practices can lower reengineering and maintenance costs and help avoid costly redesign problems. Providing multilingual services to monolingual subsystems permits incremental improvements while at the same time lowers costs through increased reuse. Finally, the registration of synonyms, user preferences, locales, and services in a global name service makes the system cohesive.

ACKNOWLEDGMENTS

I wish to thank Bob Ayers (Adobe), Joseph Bosurgi (Univel), Asmus Freytag (Microsoft), Jim Gray (Digital), and Jan te Kiefte (Digital) for their helpful comments on earlier drafts. A special thanks to Digital's internationalization team, whose contributions are always understated. In addition, I would like to acknowledge the Unicode Technical Committee, whose impact on the industry is profound and growing; I have learned a great deal from following the work of this committee.

REFERENCES

1. D. Carter, *Writing Localizable Software for the Macintosh* (Reading, MA: Addison-Wesley, 1991).
2. *Producing International Products* (Maynard, MA: Digital Equipment Corporation, 1989). This internal document is unavailable to external readers.
3. *Digital Guide to Developing International Software* (Burlington, MA: Digital Press, 1991).
4. S. Martin, "Internationalization Made Easy," OSF White Paper (Cambridge, MA: Open Software Foundation, Inc., 1991).
5. S. Snyder et al., "Internationalization in the OSF DCE -- A Framework," May 1991. This document was an electronic mail message transmitted on the Internet.
6. *X/Open Portability Guide, Issue 3* (Reading, U.K.: X/Open Company Ltd., 1989).
7. *X/Open Internationalization Guide, Draft 4.3* (Reading, U.K.: X/Open Company Ltd., October 1990).
8. *UNIX System V Release 4, Multi-National Language Supplement (MNLS) Product Overview* (Japan: American Telephone and Telegraph, 1990).

9. Information Technology -- Universal Coded Character Set (UCS) Draft International Standard, ISO/IEC 10646 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).
10. A. Nakanishi, *Writing Systems of the World*, third printing (Rutland, Vermont, and Tokyo, Japan: Charles E. Tuttle Company, 1988).
11. The Unicode Consortium, *The Unicode Standard -- Worldwide Character Encoding, Version 1.0, Volume 1* (Reading, MA: Addison-Wesley, 1991).
12. R. Haentjens, "The Ordering of Universal Character Strings," *Digital Technical Journal*, vol. 5, no. 3 (Summer 1993): 43-52.
13. *Programming Languages -- C*, ISO/IEC 9899:1990(E) (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).
14. S. Martin and M. Mori, *Internationalization in OSF/1 Release 1.1* (Cambridge, MA: Open Software Foundation, Inc., 1992).
15. J. Becker, "Multilingual Word Processing," *Scientific American*, vol. 251, no. 1 (July 1984): 96-107.
16. *Coded Character Sets for Text Communication, Parts 1 and 2*, ISO/IEC 6937 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1983).
17. J. Bettels and F. Bishop, "Unicode: A Universal Character Code," *Digital Technical Journal*, vol. 5, no. 3 (Summer 1993): 21-31.
18. Go Computer Corporation, "Compaction Techniques," *Second Unicode Implementors' Conference* (1992).
19. J. Becker, "Re: Updated [Problems with] Unbound (Open) Repertoire Paper" (January 18, 1991). This electronic mail message was sent to the Unicode mailing list.
20. V. Joloboff and W. McMahon, *X Window System, Version 11, Input Method Specification, Public Review Draft* (Cambridge, MA: Massachusetts Institute of Technology, 1990).
21. M. Davis, (Taligent) correspondence to the Unicode Technical Committee, 1992.
22. M. Gasser et al., "Digital Distributed Security Architecture" (Maynard, MA: Digital Equipment Corporation, 1988). This internal document is unavailable to external readers.
23. H. Custer, *Inside Windows NT* (Redmond, WA: Microsoft Press, 1992).

TRADEMARKS

Digital and TeamRoute are trademarks of Digital Equipment Corporation.

PostScript is a registered trademark of Adobe Systems Inc.

Windows NT is a trademark of Microsoft Corporation.

BIOGRAPHY

Gayn B. Winters Corporate consulting engineer Gayn Winters has 25 years' experience developing compilers, operating systems, distributed systems, and PC software and hardware. He joined Digital in 1984 and managed the DECmate, Rainbow, VAXmate, and PC integration architecture. He was appointed Technical Director for Software in 1989 and contributes to the Corporate software strategy. From 1990 to 1992, Gayn led the internationalization systems architecture effort and is on the Board of Directors for Unicode, Inc. He has a B.S. from the University of California at Berkeley and a Ph.D. from MIT.

=====
Copyright 1993 Digital Equipment Corporation. Forwarding and copying of this article is permitted for personal and educational purposes without fee provided that Digital Equipment Corporation's copyright is retained with the article and that the content is not modified. This article is not to be distributed for commercial advantage. Abstracting with credit of Digital Equipment Corporation's authorship is permitted. All rights reserved.
=====