TABLE OF CONTENTS

INTRODUCTION ................................................................................................................. 3
WHAT’S THE PROBLEM? ................................................................................................. 3
WHAT IS AN ACE? .......................................................................................................... 4
ACE MORPHOLOGY ......................................................................................................... 4
IDENTIFIERS .................................................................................................................. 5
SCALAR ENCODING ....................................................................................................... 5
RACE (ROW-BASED ACE) .............................................................................................. 6
LACE (LENGTH-BASED ACE) ......................................................................................... 7
DUDE (DIFFERENTIAL UNICODE DOMAIN ENCODING) .............................................. 10
WEBMETHODS ACE: WHAT ELSE ARE ACES GOOD FOR? ........................................... 12
AMC-ACE-Z OR PUNYCODE ........................................................................................ 13
WMACE REVISITED ...................................................................................................... 14
CONCLUSION ................................................................................................................ 15
INTRODUCTION

Unicode, the Universal Character Set, is a remarkable success story. Its adoption has allowed the development of multilingual software and been an important part of the success of such technologies as XML, Java, the Web, and many others.

Because Unicode encodes nearly all of the world's writing systems, developers are constantly seeking ways to use it in implementations and to solve problems.

At the same time, many developers feel the urge to tinker with the bits and build new Unicode encodings that seem to solve new problems. There are dozens of encodings that have been proposed and several new ones are proposed each year.

One particular implementation problem has resulted in a wide array of new encoding proposals. Unlike most encoding proposals in the last few years, this particular problem seems to promise widespread adoption of the results.

The "problem"? Internationalized Domain Names (IDN) on the Internet.

What's the Problem?

Existing domain names, like "webmethods.com" or "google.com" are composed entirely of 7-bit ASCII printable characters. Upper and lower case letters are considered the same, so the total number of different characters that can be used in a domain name is 37.

By contrast, the Unicode character set includes space for about 1.1 million characters (and actually encodes about 95,000 currently). Most of these characters are from writing systems used by languages other than English. There is a lot of interest, therefore, in providing domain names that include the additional characters from Unicode.

When you type a domain name into your Web browser, the computer sends a request to a special "name server" that looks up the corresponding IP address of the machine you are sending the request to. There are millions of installations of this software. Replacing them all to allow for Unicode characters in domain names while keeping the Internet running simply isn't possible.

As a result, the developers working on adding support for names like "日本語.com" and "français.net" devised a simple solution. They decided that the Unicode name would be encoded in a way that could be understood by existing name resolution software. An additional translation layer can easily be added to products like browsers to allow support for these names without requiring massive re-plumbing of the Internet host system.

Many developers rushed off to design and try out different encodings that could meet the requirements of internationalized domain names. These encodings are all similar in that they all use just the limited range of characters available in domain names, but they vary in performance and design. As a group, these encodings are called "ACEs" (for ASCII Compatible Encodings).

Like many programmatic endeavors, ACEs feature some of the most innovative and hilarious acronyms and names anywhere, with oddities like RACE, MACE, DUDE, and Punycode. In this paper we'll meet them all and look at the general approaches that were
tried, the performance of each, and winners and losers among these specialized encodings.

**What is an ACE?**

ASCII Compatible Encodings are generally similar to other Transfer Encoding Syntaxes on the Internet (see Unicode Technical Report #17). A transfer encoding syntax is defined as "a reversible transform of encoded data which may (or may not) include textual data represented in one or more character encoding schemes."

The most familiar transfer encodings are MIME, quoted-printable, and base64. ACEs borrow some of the ideas from these encodings, especially base64, and adapt them to the peculiar requirements of domain names:

- Domain names are separated into blocks of text by the "." character.
- Domain names have a fixed upper length in octets (bytes).
- Domain names consider UPPER and lower case letters to be the same.
- Domain names allow only these characters: [a-zA-Z0-9-]

The most important requirement is that the encoded sequence and its source string have a tightly bound, one-to-one relationship. That is, for any source string \( A \) there must be one and only one encoded sequence \( A' \). Similarly, for any encoded sequence \( A' \) there must be one and only one source string that can be decoded from it and that string must be identical to the original string encoded.

In order to improve the matching of source and encoded strings and reduce the ambiguity in ownership of domain names introduced by certain Unicode features (specifically combining and compatibility characters), an important feature of the proposed IDN solution is a normalization process called NAMEPREP. Since this paper deals with the technical details of ACE encodings, I will ignore NAMEPREP here but implementers should be aware of its existence as a consideration for IDN.

**ACE Morphology**

As IDN evolved, each new ACE built on the lessons of the previous proposal. Each encoding has its own strengths and weaknesses. The basic evolution is described by the general classifications of the encodings:

1. Direct Unicode Scalar Value encoding. [LACE, RACE]
2. Differential encoding of scalar values. [DUDE]
3. Differential encoding of scalar values with code point reordering. [AMC-ACE-Z, a.k.a. Punycode]

In addition to the foregoing, we'll examine webMethods experimentation with DUDE and Punycode in Java compiler technology.

1 [http://www.unicode.org/reports/tr17](http://www.unicode.org/reports/tr17)
Identifiers

As a brief aside, note that existing domain names must continue to function after the introduction of iDNS names. All IDN names, therefore, include a signature string. For example, LACE names are always prefixed with the string "lq--", while DUDE names are prefixed with "dq--". In this paper, I will note a signature associated with each encoding, but will omit the signatures from the examples for purposes of brevity.

Scalar Encoding

Thirty-seven distinct values works about to be (a little more than) five bits of data. A Unicode Scalar Value is 21 bits long. The easiest mapping would just assign the letters a through z and the numbers zero through nine as binary values and directly encode the scalar value.

If this sounds familiar, it's because there is already a transfer encoding in common use that uses this feature: it's Base64. Base64 has the advantage of being able to use all of printable ASCII and in recognizing the distinction between upper and lower case.

<table>
<thead>
<tr>
<th>bits</th>
<th>char</th>
<th>hex</th>
<th>bits</th>
<th>char</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>a</td>
<td>0x61</td>
<td>00000</td>
<td>q</td>
<td>0x71</td>
</tr>
<tr>
<td>00001</td>
<td>b</td>
<td>0x62</td>
<td>00010</td>
<td>c</td>
<td>0x63</td>
</tr>
<tr>
<td>00010</td>
<td>d</td>
<td>0x64</td>
<td>00011</td>
<td>d</td>
<td>0x65</td>
</tr>
<tr>
<td>00100</td>
<td>e</td>
<td>0x66</td>
<td>00101</td>
<td>f</td>
<td>0x67</td>
</tr>
<tr>
<td>00110</td>
<td>g</td>
<td>0x68</td>
<td>00111</td>
<td>h</td>
<td>0x69</td>
</tr>
<tr>
<td>01000</td>
<td>i</td>
<td>0x6a</td>
<td>01001</td>
<td>j</td>
<td>0x6b</td>
</tr>
<tr>
<td>01010</td>
<td>k</td>
<td>0x6c</td>
<td>01011</td>
<td>l</td>
<td>0x6d</td>
</tr>
<tr>
<td>01100</td>
<td>m</td>
<td>0x6e</td>
<td>01101</td>
<td>n</td>
<td>0x6f</td>
</tr>
<tr>
<td>01110</td>
<td>o</td>
<td>0x70</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All ACEs use some variation of this "Base32" encoding. You'll note the above version doesn't use the numbers '0' and '1' (in order to avoid human transcription problems due to confusion of zero and the letter 'o' and one and the letter 'L').

Here's how it works. Take the input sequence ユニコード. That's <U+30E6 U+30CB U+30B3 U+30FC U+30C9>. If we encode the 21-bit scalar values using the above chart to encode this sequence we get: "aamhgaamglamftaamaamh4aamgi"
With each character representing five bits, each Unicode Scalar Value requires five bytes to encode it. This is enormously wasteful. Since DNS names have a fixed upper limit in size, this encoding would restrict non-ASCII names to a minuscule size. This isn't strictly necessary, especially since very few Unicode characters use the upper five bits of the scalar value, and it resulted in the first search for improved encodings.

One obvious improvement is to encoding the UTF-16 code point values instead of the scalar values. Many ACE encodings use this technique to reduce the encoded byte count on Basic Multilingual Plane characters (that vast majority of characters in common use), at the cost of greatly increasing the byte count required to encoded supplemental plane characters. For example, the same sequence is encoded as: "amhgamglamftamh4amgj". However, the sequence <U+10FFFE U+10FFFF> (just two characters) is "bw77bx77bw77bw76", or almost the same length as our sequence of five BMP characters. Note that every surrogate pair requires eight bytes in this format, while BMP characters require just four.

**RACE (Row-based ACE)**

One of the first improvements tried was the row-based encoding proposal of Paul Hoffman. This encoding worked exactly as described above, but takes advantage of the fact that Unicode is divided into blocks. The blocks are usually distinguished by the writing system used. A meaningful DNS name will generally be a word that uses a single writing system, possibly mixed with ASCII values.

You can see this in the repetitive nature of the example above. If we could chop off the repetitive "aam" portion of the encoded sequence we would save 18 bytes!

RACE takes advantage of this by encoding the "high byte" of each UTF-16 code point as the first value in its sequence and then just the "low bytes" of each character. Of the 35 Unicode blocks named for a single writing system, only six span more than a single "row" (or high byte).
Here's how RACE works:

1. Turn the name into a UTF-16 sequence. Characters outside the Basic Multilingual Plane (that is, with scalar values larger than 0xFFFF) are encoded as a surrogate pair sequence.

2. Scan the high bytes of each character. If they are all the same or if they are all the same, but mixed with some "0x00" bytes (that is, there is ASCII mixed in), then RACE encode the sequence, otherwise encode the byte 0xD8 and then encode the sequence as above.

3. The "common" high byte is encoded as the first in the sequence. The each low byte is encoded, one after another. If an ASCII character is in the sequence, it is "escaped" by encoding 0xFF. If the low byte is 0xFF, it is encoded as 0xFF 0x99.

Here's our example:

<table>
<thead>
<tr>
<th>Base32</th>
<th>RACE</th>
<th>LACE</th>
<th>DUDE</th>
<th>wmACE</th>
<th>PyrG</th>
</tr>
</thead>
<tbody>
<tr>
<td>RACE</td>
<td>有限 □</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence:</td>
<td>0x303E0x30CB0x303E0x30FC0x30C9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoded:</td>
<td>bq-GDTMkM74E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decoded:</td>
<td>有限 □</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Length:</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoded Length:</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td>2.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RACE is quite efficient for sequences that meet its criteria. However, it is important to note that Chinese, Japanese, and Korean writing systems span many dozens of "rows" in Unicode and that many of symbols and accented or presentation characters in Latin-based languages (like English), Arabic, and Greek scripts are located in disparate rows. For these RACE is only as efficient as the original Base32 encoding examples.

RACE STATISTICS
Signature: bq--
Best efficiency (single row, 17 characters): 2.12 bytes per character
Worst efficiency (8 surrogates, 1 non-ASCII): 7.56 bytes
Described: http://www.i-d-n.net/draft/draft-ietf-idn-race-03.txt

LACE (Length-based ACE)
Mark Davis and Paul Hoffman collaborated to improve the design of RACE by creating LACE.

LACE borrows the idea of encoding the high byte in each UTF-16 code point and the idea that most character will come from a single Unicode block. Unlike RACE, LACE encodes the number of high bytes in sequence, so it can "switch blocks". In some ways, LACE is
similar to the SCSU ("Standard Compression Scheme for Unicode") encoding, in that is tries to eliminate redundant bytes from the output stream.

Here's the basic idea:

1. Count the number of continuous UTF-16 code points that share the same high-byte.
2. Encode the number of bytes in Base32.
3. Encode the high byte.
4. Encode each of the low bytes.
5. Go to 1 until you run out of string.
6. If the output sequence is longer than the original string (that is, if you didn't compress the string by using LACE), encoding 0xFF and then the original, uncompressed string.

Let's look at our example string again:

\[<U+012F U+0111 U+0149 U+04E5 U+0432>\]

It's longer than the RACE example! This is because the number of characters and the lead byte (05 30) have to be encoded (10 bits are required).

Where LACE beats RACE, though, is when the high-byte changes. Let's use the following example, which mixes Cyrillic and Latin script characters:

\[<U+012F U+0111 U+0149 U+04E5 U+0432>\]
Four ACEs: A Survey of ASCII Compatible Encodings

LACE has, therefore, several advantages over RACE. It's still weak with surrogates, but it is a robust, fairly compact encoding. For awhile it seemed like the winner…

LACE STATISTICS
Signature: lq--
Best efficiency (single row, 17 characters):
Best efficiency (7 characters row 2, 10 characters row 3):
Worst efficiency (8 surrogates, 1 other):
Described: http://www.i-d-n.net/draft/draft-ietf-idn-lace-01.txt
DUDE (Differential Unicode Domain Encoding)

While LACE was pretty good, an alternate approach turned out to be more efficient in a couple of ways. DUDE was originally proposed by Mark Welter, Brian Spolarich, and Adam Costello and it features a different approach to encoding Scalar Values. [One wonders how late the authors stayed up thinking of the acronym.]

In all of the foregoing examples, each character or UTF-16 code point is directly encoded using Base32. LACE and RACE make use of the idea that characters in a writing system are generally grouped together in a single Unicode block and thus the high byte in their UTF-16 code points are usually redundant.

The problem is that, while many "names" are likely to come from a specific Unicode block, many Latin languages use several blocks. And Far East Asian languages like Japanese and Chinese (with large computer-using populations) have blocks that span dozens of rows (that is, the high byte isn't the same).

DUDE differs from the encodings that come before because it encodes the difference between code points (via XOR), rather than directly encoding the specific scalar value—hence the name "differential". (Note that the Base32 characters selected for DUDE differ slightly from those in LACE/RACE and Table 1, although the use of lower case in the demo program has nothing to do with this—rather it is an artifact of the particular DUDE implementation used).
Here is the pseudo-code from the proposal:

```plaintext
let prev = 0x60
for each input integer n (in order) do begin
  if n == 0x2D then output hyphen-minus
  else begin
    let diff = prev XOR n
    represent diff in base 16 as a sequence of quartets,
    as few as are sufficient (but at least one)
    prepend 0 to the last quartet and 1 to each of the others
    output a base-32 character corresponding to each quintet
    let prev = n
  end
end
```

The base value of 0x60 is slightly more efficient than starting at 0x00.

It's important to note the "prepend 0..." part. DUDE is a highly patterned encoding. It is easy to find character boundaries using bit logic and should be relatively self-correcting.

LACE is limited to a maximum of 32 characters of non-ASCII input. RACE has no built in limit, but the longer a non-ASCII string is, the less likely it is to be RACE encoded (with any benefit).

DUDE, by contrast, can be used as a general-purpose transfer encoding. It has no built in size limit and is streamable because each character's encoding only depends on the previous character state. LACE and RACE both require the encoder to "look ahead" in the string buffer (RACE actually requires the buffer to be read fully at least twice).

It also maintains its relative efficiency regardless of the characters chosen to encode. The perverse case within the BMP requires four bytes per character to encode the sequence `<U+FFFD U+0020>` (993p997p), which is the same as LACE or RACE. The sequence `<U+10FFFD U+0020>` requires just 11 bytes to encode (753rwsa797r), compared to 16 for RACE (and 17 for LACE).

The same statistical distribution rules that made RACE and LACE initially appealing apply to DUDE as well: characters in domain names are likely to be close to one another in Unicode. Unlike LACE or RACE, "close" is not bound to a specific 255 character "row" of Unicode. Where DUDE gains in efficiency compared to these earlier encodings is with Far East Asian languages, which have large Unicode blocks. In a comparison done by Chanki Park and KyungJae Park\(^2\), DUDE averaged around 60 bytes per 30 character Hangul sequence (compared to more than 70 for LACE).

Because DUDE was not a direct improvement of the same encoding scheme, it is not always more efficient than LACE, so direct comparisons on efficiency require simulation of the likely distribution of domain names, especially for the large Far East Asian scripts. DUDE's performance advantage here made it a desirable encoding.

Finally, because it reads the buffer just once and its performance is tied to a bitwise operator, DUDE has generally got higher performance compared to the other algorithms presented here.

---

\(^2\) [http://www.i-d-n.net/draft/draft-park-idn-ace-eval-kr-00.txt](http://www.i-d-n.net/draft/draft-park-idn-ace-eval-kr-00.txt)
webMethods ACE: What Else are ACEs Good For?

Around the time that DUDE was being considered for IDN, I became interested in the use of ACEs for another purpose. My problem required these attributes:

- Unicode names (of limited upper length) must be stored in an ASCII-only format.
- The names must be decoded reliably.
- Strings must encode the same way each time.
- The encoded string must be unique.

These requirements are remarkably similar to those of IDN. Not wanting to invent a new encoding of my own, I investigated the progress made on ACEs by the IDN working group.

The problem? One webMethods product generates Java source code from a drawing of a business process or human-interactive workflow. The name the user gives to the process or process step ideally is the name given to the class file.

Java's requirements are actually quite lenient in this regard. Within certain restrictions the product could generate the code and compile the classes successfully. The problem comes from the Java class loader, which is reliant on the local filesystem. Ideally the class filename would not be dependent on the local filesystem and it should be as efficient as possible.

The first pilot project adopted a variation of the DUDE encoding. I was able to create a Java ClassLoader that loaded classes given their (unencoded) name and I was able to generate Java source code that used the encoded names as class names reliably. The previous implementation had encoded the name as four ASCII bytes representing the UTF-16 byte value (e.g. U+20AC encoded as "20ac"). My adaptation allowed for ASCII readability by introducing a "shift-state" between pure ASCII and DUDE encoding.
In many ways, "wMACE" was very appealing. It was more efficient than Java's escaping mechanism (\uXXXX) and was quite portable. The algorithm was reasonably efficient. Fellow developers were amused by both of the acronyms.

DUDE was not, however, the last word in ACE performance.

**AMC-ACE-Z or Punycode**

The current "last word" in ACE performance is an encoding with the bizarre name "Punycode". The encoding was originally called AMC-ACE-Z, because it is the 26th in the series by its author. The "AMC" is the encoding author's initials (Adam M. Costello).

Punycode is based on the same general idea as DUDE, which is the differential encoding of characters in a domain name. However, because of the way that Punycode works, the bit manipulation is much more complex than a simple XOR. This is because Punycode adopts two additional mechanisms for even greater efficiency.

The first additional trick is code point reordering.
With DUDE we performed XOR operations to get the bits to encode. This reduces the number of bits to be encoded when the characters are relatively close together within Unicode. Punycode's expansion on this is to recognize that the greatest efficiency arises if the characters in the string are sorted into code point order.

For example, the string "米国" (Japanese for "USA") is composed of the Unicode characters U+7C73 U+56FD. In DUDE, these code points are encoded in their order in the string: " z6tdu42q". In Punycode they are encoded in the most efficient order, which is reversed: " -jgsr58f", for a savings of one encoded byte. The encoding sequence includes the position to insert the decoded character into the string.

The other trick that Punycode introduces is that ASCII sequences are not encoded. This means two things:

1. ASCII characters in the middle of the input stream don't affect the encoder's efficiency, they way that they do in DUDE. Since ASCII characters are the most likely to be inserted in the middle of a sequence in another Unicode block, this delivers a good bit of real world efficiency.

2. ASCII sequences are human readable.

Consider the sequence "安室奈美恵-with-super-monkeys".

DUDE encodes this as the inscrutable:

```
 dq→x58jupu8nuy6gt39myxrikhtdpnhvbcqfttfrwacbfqtmk
```

Punycode encodes this as:

```
 withsupermonkeys-4vaf22876bg5ia8qsi0cg7n9n
```

So Punycode is not just more efficient, it's a little friendlier to boot.

Studies of Punycode indicate a very small difference for most scripts. However, for real CJK domain names it does provide both a slight improvement in efficiency and moderate usability improvements. Efficiency improvement is an advantage, because improved encoding efficiency translates into longer allowable domain names, which in turn means greater flexibility and openness for non-ASCII domains.

**PUNYCODE STATISTICS**

*Signature: pq--*


**wMACE Revisited**

Once Punycode was enshrined as the "King of ACEs", I wrote an implementation to replace my previous DUDE encoder. Since Punycode preserves ASCII readability just like the previous wMACE, I was able to implement the encoding more efficiently.

wMACE solved webMethods problems with code generation and the latest generation of webMethods Workflow uses Punycode to generate some Java .class file names in a multilingual environment.
Conclusion

Ultimately the English-centric past of the Internet will fade. The vicissitudes of creating an IETF standard have delayed the adoption of an IDN standard for now, and the demise of RealNames as a working company (they were maintaining actual domains encoded as LACE as a plug-in bundled with Internet Explorer) has also dealt a setback to internationalized domain names. Nonetheless, there are still interesting applications for ASCII Compatible Encodings and no doubt IDN in some form will be adopted. In the meantime, exploring these interesting encodings has led to a better understanding of character distribution and encoding performance overall, as well as providing some useful tools for ASCII-only environments.