Internationalizing Mail Addresses in Applications (IMAA)

Abstract

The Internationalizing Domain Names in Applications (IDNA) specification describes how to process domain names that have characters outside the ASCII repertoire. A user who has an internationalized domain name may want to have their full Internet mail address internationalized, including the local part (that is, the part to the left of the "@"). This document describes how to use non-ASCII characters in local parts, by defining internationalized local parts (ILPs), internationalized mail addresses (IMAs), and a mechanism called IMAA for handling them in a standard fashion.

1. Introduction

A mail address consists of local part, an at-sign (@), and a domain name. The IDNA specification [IDNA] describes how to handle domain names that have non-ASCII characters. This document describes how to handle non-ASCII characters in the rest of the mail address.

This document explicitly does not discuss internationalization of display names and comments in mail addresses that appear in message headers [RFC2822]. MIME part three [RFC2047] describes how use an extended set of characters in message headers, and this document does not alter that specification.

This document is being discussed on the ietf-imaa mailing list. See <http://www.imc.org/ietf-imaa/> for information about subscribing and the list’s archive.
1.1 Relationship to IDNA

This document relies heavily on IDNA for both its concepts and its justification. This document omits a great deal of the justification and design information that might otherwise be found here because it is identical to that in IDNA. Anyone reading this document needs to have first read [IDNA], [PUNYCODE], [NAMEPREP], and [STRINGPREP].

The main differences between how IMAA treats local parts of mail addresses and how IDNA treats domain names are:

- The ACE prefix for internationalized local parts is different from the ACE prefix for internationalized domain labels.

  [[ OPEN ISSUE: Should it be the same? ]]

- Domain names have an intrinsic segmentation into labels, and are already segmented before transformations are performed. Local parts, on the other hand, have no intrinsic segmentation. The transformations on local parts perform a segmentation internally, but it has no external significance.

- There is no UseSTD3ASCIIIRules flag for local parts.

One apparent difference that is not really a difference is the handling of quoting mechanisms. IDNA did not discuss quoting because the phrase "domain label" is presumed to refer to a simple literal string. [STD13] defines domain labels in terms of their literal form (which is used in DNS protocol messages), and later introduces a quoting syntax for representing domain labels in master files, but there is never any doubt that the domain label itself is a simple unstructured sequence. It goes without saying that domain labels obtained from contexts that use quoting (like master files) need to be reduced to their literal form before any processing is done on them.

Local parts, on the other hand, are defined in [RFC2822] and [RFC2821] in terms of their quoted form, as they appear in message headers and SMTP commands. Later it is stated that the quotation characters are not really part of the local part. To avoid any ambiguity, IMAA explicitly discusses the process of dequoting and requoting local parts.

1.2 Open issues

This section describes the issues that are known to be unresolved. There may also be other issues we haven’t thought of yet. This section might be easier to follow after the rest of the draft has been read. This section will be removed before the document is passed to the IESG or RFC Editor for publication.

Throughout the draft, comments related to these open issues appear inside brackets like this: [[ OPEN ISSUE: comments ]].

The IMAA model in this draft is incompatible with case-sensitive mail exchangers, and therefore IMAs cannot be created in domains whose mail exchangers are case-sensitive. Case-sensitivity in mail exchangers is allowed but discouraged by [RFC2821], and is thought to be very rare. It would be possible for IMAA to
support case-sensitive mail exchangers, but it would entail complications to the model. Non-traditional local parts would not always be case-insensitive, but could be either case-insensitive or lowestcase-only (the concept of lowestcase would need to be defined). Instead of the symmetric notion of "equivalence" between local parts, there would be an asymmetric notion of "substitutability" (whose definition would depend on the concept of lowestcase). The ToASCII and ToUnicode operations would be constrained to preserve the lowestcase property (that is, the output must be lowestcase if the input is lowestcase). The details have all been worked out, but perhaps it is not worth the trouble, and better to just let case-sensitive mail exchangers go unsupported.

Currently hyphen is not a protected character, because it is used by both Punycode and the ACE prefix. It is possible, however, to avoid the use of hyphen for those purposes, which would allow hyphen to be protected, for better compatibility with structured local part conventions that use hyphen as a delimiter. Here is how it could be done: After applying the Punycode encoder, instead of prepending the ACE prefix, insert the ACE infix in place of the hyphen (or prepend the infix if there is no hyphen). On the decoding side, instead of looking for the ACE prefix and removing it, look for the ACE infix and change it to a hyphen (or just delete it if it occurs at the beginning), then apply the Punycode decoder.

If we decide to stick with a prefix containing hyphens, we might want to consider reusing the IDNA ACE prefix (this was not considered in draft 00 because in that draft IMAA used a different stringprep profile from IDNA). The disadvantage of using a different prefix is that humans cannot, without computational assistance, copy local parts into domain labels (as in SOA records) or copy domain names into local parts, because copying the non-ASCII form and then converting to ASCII would give a different result versus converting to ASCII and then copying, and it’s the latter procedure that must be considered correct (for compatibility with IMA-unaware and IDN-unaware software that might try to do the same sort of copying). Furthermore, once the copying has happened, the result will display unintelligibly (the ACE will be visible), because the different ACE prefix won’t be recognized on the other side of the at-sign. It is impossible to fully solve this problem, because encoded strings don’t mark their own endings, only their own beginnings. Even if the same ACE prefix is used on both sides of the at-sign, if local parts are segmented then a multi-segment local part copied into a domain label will not display intelligibly, while if local parts are not segmented then a multi-label domain name copied into a local part will not display intelligibly. However, using the same ACE prefix would allow the common cases to work intuitively: Local parts containing only LDH characters and non-ASCII characters could be copied (by humans, in non-ACE form) into domain labels (where they would display correctly), and domain names obeying the STD3 ASCII rules could be copied (by humans, in non-ACE form) into local parts (where they would display correctly). One concern with using the same prefix is that in the uncommon cases where it doesn’t work nicely, the unintelligible display will not be an ACE, but will be non-ASCII gobbledygook (which will still work if copied back to the other side of the at-sign, but might be even less user-friendly than an ACE).

Should we keep the requirement about recognizing fullwidth at-signs? It seems needed for consistency with IDNA’s requirement about recognizing fullwidth dots.
If we were to drop the at-sign requirement, it would become possible to narrow our focus from "mail address slots" to "local part slots". But would we want to do that? If we keep the at-sign requirement, it's a moot point, because then we're talking about the whole address.

When converting mail addresses to ASCII, should ideographic full stop be converted to ASCII full stop in local parts, as is done in domain names? This was desirable in domain names because all domain names contain dots, so we wanted them to be easy to type. But local parts need not contain dots, and most don’t, so that motivation is not nearly as compelling in local parts. Also, the conversion in IDNA makes it difficult or impossible to include ideographic full stop inside domain labels. If the conversion were done in local parts, the same difficulty would arise. Users might prefer the ability to use honest-to-goodness ideographic full stops in local parts, rather than reserve them as a typing shortcut for ASCII full stops. For example, one of the most well-known pop groups in Japan, Morning Musume, has an ideographic full stop in their name.

In the dequoting step, fullwidth versions of nonliteral ASCII characters (like quote marks and backslashes) are required to be recognized as equivalent to the regular ASCII versions. Should we keep this requirement?

In the requoting step, the original quoted local part is recommended when ToASCII/ToUnicode had no effect and the original quoting style is compatible with the destination context. Should we keep that recommendation? It adds complexity, and should not be necessary, but it makes IMAA less likely to trigger quotation-related bugs, and is motivated by the principle of not altering local parts unnecessarily (for example, when converting an already-ASCII local part to ASCII, don’t gratuitously change the way it’s quoted).

The 59-character limit on the Punycode encoder output is aimed at making it easier to reuse Punycode implementations that were written for IDNA (and which might use fixed-sized buffers). Should this limit be relaxed for IMAA? Unlike domain labels, which have a hard size limit imposed by the syntax of DNS messages, local parts have no hard limit (SMTP must support local parts up to 64 character, but may support arbitrarily large local parts). A Punycode implementation using 31-bit unsigned integers (or 32-bit signed integers) ought to be able to handle Unicode strings in excess of 2000 code points (I have not calculated the exact limit). For very long strings, the \(O(n^2)\) running time of Punycode might become an issue.

What more should we say about stored strings versus query strings?

### 1.3 Closed issues that could be reopened

Rather than transform the local part as multiple segments, another approach is to transform it as a single unit. The tradeoff is complexity versus compatibility with various unofficial conventions for structured local parts, like owner-listname, user+tag, sublocal.local, path!user, etc. Breaking a local part into segments is about as complex as breaking a domain name into labels.

If segmentation were abandoned, we would lose a major reason to avoid punctuation in the ACE prefix. By using using punctuation
other than hyphens, we could use the same letters as IDNA. For example, the IDNA ACE prefix is xn--, and the IMAA ACE prefix could be xn__.

2. Terminology

The key words "MUST", "SHALL", "REQUIRED", "SHOULD", "RECOMMENDED", and "MAY" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Code point, Unicode, and ASCII are defined in [IDNA].

Each ASCII character whose code point is in the range 21..7E has a corresponding "fullwidth version" whose code point is in the range FF01..FF5E, respectively.

[[ OPEN ISSUE: The above definition is not needed if the requirement about fullwidth versions of nonliteral ASCII characters is removed. ]]

The "protected code points" are 0..40, 5B..60, 7B..7F (in other words, those corresponding to ASCII characters other than letters, digits, and hyphen-minus).

[[ OPEN ISSUE: We might want to add hyphen-minus to the set of protected characters, but we’d need to deal with the use of hyphen-minus by Punycode and the ACE prefix. ]]

A "mail address" consists of a local part, an at-sign, and a domain name, in that order. The exact details of the syntax depend on the context; for example, a "mailbox" in [RFC2821] (SMTP) and an "addr-spec" in [RFC2822] (message format) are both mail addresses, but they define slightly different syntaxes for local parts and domain names.

A "dequoted local part" is the simple literal text string that is the intended "meaning" of a local part after it has undergone lexical interpretation. A dequoted local part excludes optional white space, comments, and lexical metacharacters (like backslashes and quotation marks used to quote other characters). Dequoted local parts are generally not allowed in protocols (like SMTP commands and message headers), but they are needed by IMAA as an intermediate form. The dequoted form of X is sometimes written dequote(X).

An "internationalized local part" (ILP) is anything that satisfies both of the following conditions: (1) It conforms to the same syntax as a non-internationalized local part except that (a) non-ASCII Unicode characters are allowed wherever ASCII letters are allowed, and (b) for every ASCII character that has a nonliteral meaning (like quotation or comment delimitation), the fullwidth version (if there is one) has the same meaning. (2) After it has been dequoted, the ToASCII operation can be applied to it without failing (see section 4). The term "internationalized local part" is a generalization, embracing both old ASCII local parts and new non-ASCII local parts. Although most Unicode characters can appear in internationalized local parts, ToASCII will fail for some inputs. Anything that fails to satisfy condition 2 is not a valid internationalized local part.

[[ OPEN ISSUE: Should we keep (1)(b)? ]]
A "traditional local part" is a local part that contains only ASCII characters and whose dequoted form would be left unchanged by the ToUnicode operation (see section 4).

An "internationalized mail address" (IMA) consists of an internationalized local part, an at-sign, and an internationalized domain name [IDNA], in that order.

Equivalence of local parts is defined in terms of the dequoted form (see above) and the ToASCII operation, which constructs an ASCII form for a given dequoted local part (whether or not the local part was already an ASCII local part). Two traditional local parts X and Y are equivalent if and only if dequote(X) and dequote(Y) are exactly identical. (That is not a new rule, it is inferred from [RFC2821] and [RFC2822].) For internationalized local parts X and Y that are not both traditional, they are defined to be equivalent if and only if ToASCII(dequote(X)) matches ToASCII(dequote(Y)) using a case-insensitive ASCII comparison. Unlike traditional local parts, non-traditional internationalized local parts are always case-insensitive.

Two internationalized mail addresses are equivalent if and only if their local parts are equivalent (according to the previous definition) and their domain parts are equivalent (according to IDNA).

To allow internationalized labels to be handled by existing applications, IDNA uses an "ACE local part" (ACE stands for ASCII Compatible Encoding). An ACE local part is an internationalized local part that can be rendered in ASCII and is equivalent to an internationalized local part that cannot be rendered in ASCII. Given any internationalized local part (in dequoted form) that cannot be rendered in ASCII, the ToASCII operation will convert it to an equivalent ACE local part (whereas an ASCII local part will be left unaltered by ToASCII). ACE local parts are unsuitable for display to users. The ToUnicode operation will convert any local part (in dequoted form) to an equivalent non-ACE local part. In fact, an ACE local part is formally defined to be any local part that the ToUnicode operation would alter (whereas non-ACE local part are left unaltered by ToUnicode). The ToASCII and ToUnicode operations are specified in section 4.

The "ACE prefix for local parts" (or simply the "ACE prefix" when the context is clear) is defined in this document to be a string of ASCII characters that begins every encoded segment within a dequoted ACE local part. It is specified in section 5.

[ [ OPEN ISSUE: It might be preferrable to use an infix rather than a prefix. ] ]

A "mail address slot" is defined in this document to be a protocol element or a function argument or a return value (and so on) explicitly designated for carrying a mail address. Mail address slots exist, for example, in the MAIL and RCPT commands of the SMTP protocol, in the To: and Received: fields of message headers, and in a mailto: URI in the href attribute of an HTML <A> tag. General text that just happens to contain an mail address is not a mail address slot; for example, a mail address appearing in the plain text body of a message is not occupying a mail address slot.
An "IMA-aware mail address slot" is defined in this document to be a mail address slot explicitly designated for carrying an internationalized mail address as defined in this document. The designation may be static (for example, in the specification of the protocol or interface) or dynamic (for example, as a result of negotiation in an interactive session).

An "IMA-unaware mail address slot" is defined in this document to be any mail address slot that is not an IMA-aware mail address slot. Obviously, this includes any mail address slot whose specification predates this document.

3. Requirements and applicability

3.1 Requirements

IMAA conformance means adherence to the following four requirements:

1) In an internationalized mail address, the following characters MUST be recognized as at-signs for separating the local part from the domain name: U+0040 (commercial at), U+FF20 (fullwidth commercial at).

   [[[ OPEN ISSUE: Keep that requirement? ]]]

2) Whenever a mail address is put into an IMA-unaware mail address slot (see section 2), it MUST contain only ASCII characters. Given an internationalized mail address, an equivalent mail address satisfying this requirement can be obtained by applying ToASCII to the local part as specified in section 4, changing the at-sign to U+0040, and processing the domain name as specified in [IDNA].

3) ACE local parts obtained from mail address slots SHOULD be hidden from users when it is known that the environment can handle the non-ACE form, except when the ACE form is explicitly requested. When it is not known whether or not the environment can handle the non-ACE form, the application MAY use the non-ACE form (which might fail, such as by not being displayed properly), or it MAY use the ACE form (which will look unintelligible to the user). Given an internationalized local part, an equivalent non-ACE local part can be obtained by applying the ToUnicode operation as specified in section 4. When requirements 2 and 3 both apply, requirement 2 takes precedence.

4) If two mail addresses are equivalent and either one refers to a mailbox, then both MUST refer to the same mailbox, regardless of whether they use the same form of at-sign.

Discussion: This implies that non-ASCII local parts cannot be deployed in domains whose mail exchangers are case-sensitive. IMAA is designed to work without upgrading mail exchangers, but it works only for mail exchangers that treat ASCII local parts as case-insensitive (which is the common and preferred behavior). All local parts received by an IMA-unaware mail exchanger are ASCII, either traditional or ACE, and a case-insensitive exchanger will automatically obey requirement 4 without being aware of it. Case-sensitive exchangers will not correctly handle ACE local parts, but administrators can simply
refrain from creating ACE local parts in those domains. This is necessary because a round-trip through ToUnicode and ToASCII is not case-preserving, and therefore the result might refer to a different mailbox (in violation of requirement 4) if interpreted by a case-sensitive mail exchanger.

[[ OPEN ISSUE: IMAA could work with case-sensitive mail exchangers if we added some complexity to the model. ]]

3.2 Applicability

IMAA is applicable to all mail addresses in all mail address slots except where it is explicitly excluded.

This implies that IMAA is applicable to protocols that predate IMAA. Note that mail addresses occupying mail address slots in those protocols MUST be in ASCII form (see section 3.1, requirement 2).

3.2.1. Case-sensitive local parts

IMAA does not apply to local parts that are interpreted case-sensitively (see section 3.1 requirement 4).

4. Conversion operations

An application converts a local part put into an IMA-unaware mail address slot or displayed to a user. This section specifies the steps to perform in the conversion, and the ToASCII and ToUnicode operations.

The input to ToASCII or ToUnicode is a dequoted local part that is a sequence of Unicode code points (remember that all ASCII code points are also Unicode code points). If a local part is represented using a character set other than Unicode or US-ASCII, it will first need to be transcoded to Unicode.

Starting from a local part, the steps that an application takes to do the conversions are:

1) Decide whether the local part is a "stored string" or a "query string" as described in [STRINGPREP]. If this conversion follows the "queries" rule from [STRINGPREP], set the flag called "AllowUnassigned".

[[ OPEN ISSUE: We need more here, possibly pointing to a different section where we specify exactly what kinds of things are stored and queries. ]]

2) Save a copy of the local part.

3) Dequote the local part; that is, perform lexical interpretation and remove all nonliteral characters. For example, for local parts that use the lexical syntax of [RFC2821] (SMTP) or [RFC2822] (message format), unfold it, remove comments and unquoted white space, and remove backslashes and quotation marks used to quote other characters. The result is a simple literal text string. Fullwidth versions of nonliteral ASCII characters MUST be accepted as equivalent to the ASCII versions.

4) Process the string with either the ToASCII or the ToUnicode
operation as appropriate. Typically, you use the ToASCII operation if you are about to put the local part into an IMA-unaware slot, and you use the ToUnicode operation if you are displaying the local part to a user.

5) Apply whatever quoting is needed in the destination context (if any). For "mailbox" slots [RFC2821] and "addr-spec" slots [RFC2822] the following action suffices: If the string contains any control characters, spaces, or specials [RFC2822], or if it begins or ends with a dot, or contains two consecutive dots, then convert it to a quoted-string: insert a backslash before every quotation mark and backslash, then enclose the string with quotation marks. If step 4 had no effect on the string, and if the saved local part from step 2 is a valid representation of the string in the destination context, then the saved local part SHOULD be used, even if it uses more quoting than necessary.

[[ OPEN ISSUE: Keep that last sentence and step 2? ]]

The destination context might also impose a length restriction. Depending on whether the restriction applies to the quoted form or the dequoted form, the application might want to check the length just before or after step 5.

The following two subsections define the ToASCII and ToUnicode operations that are used in step 4.

This description of the protocol uses specific procedure names, names of flags, and so on, in order to facilitate the specification of the protocol. These names, as well as the actual steps of the procedures, are not required of an implementation. In fact, any implementation which has the same external behavior as specified in this document conforms to this specification.

4.1 ToASCII

The ToASCII operation takes a sequence of Unicode code points that make up a dequoted local part and transforms it into a sequence of code points in the ASCII range (0..7F). If ToASCII succeeds, the original sequence and the resulting sequence are equivalent dequoted local parts.

It is important to note that the ToASCII operation can fail. ToASCII fails if any step of it fails. If any step of the ToASCII operation fails, that string MUST NOT be used as an internationalized local part. The method for dealing with this failure is application-specific.

The inputs to ToASCII are a sequence of code points, and the AllowUnassigned flag. The output of ToASCII is either a sequence of ASCII code points or a failure condition.

ToASCII never alters a sequence of code points that are all in the ASCII range to begin with. Applying the ToASCII operation multiple times has exactly the same effect as applying it just once.

ToASCII consists of the following steps:

1. If the sequence contains any code points outside the ASCII range (0..7F) then proceed to step 2, otherwise stop, leaving the sequence unchanged.
2. Perform the steps specified in [NAMEPREP] and fail if there is an error. The AllowUnassigned flag is used in [NAMEPREP].

3. If the sequence is empty then stop, leaving an empty result.

4. Divide the sequence into segments. Segment boundaries occur wherever a protected code point is adjacent to a non-protected code point, and nowhere else. (Therefore segments are never empty, and they alternate between segments containing only protected code points and segments containing only non-protected code points.)

5. For each segment perform the following substeps:
   
   (a) If the segment contains any code points outside the ASCII range (0..7F) then proceed to substep b, otherwise leave the segment unchanged.
   
   (b) Verify that the segment does NOT begin with the ACE prefix.
   
   (c) Encode the sequence using the encoding algorithm in [PUNYCODE] and fail if there is an error.
   
   (d) Verify that the result contains no more than 59 code points.

   [[ OPEN ISSUE: Relax this restriction? ]]

   (e) Prepend the ACE prefix.

6. Rejoin the segments into a single sequence.

4.2 ToUnicode

The ToUnicode operation takes a sequence of Unicode code points that make up a dequoted local part and returns a sequence of Unicode code points. If the input sequence is a dequoted local part in ACE form, then the result is an equivalent dequoted internationalized local part that is not in ACE form, otherwise the original sequence is returned unaltered.

ToUnicode never fails. If any step fails, then the original input sequence is returned immediately in that step.

The ToUnicode output never contains more code points than its input. Note that the number of octets needed to represent a sequence of code points depends on the particular character encoding used.

The inputs to ToUnicode are a sequence of code points, and the AllowUnassigned flag. The output of ToUnicode is a sequence of code points.

ToUnicode consists of the following steps:

1. If the sequence contains any code points outside the ASCII range (0..7F) then proceed to step 2, otherwise skip to step 3.

2. Perform the steps specified in [NAMEPREP] and fail if there is an error. The AllowUnassigned flag is used in [NAMEPREP].
3. Verify that the sequence is nonempty, and save a copy of the sequence.

4. Divide the sequence into segments (same as step 4 of ToASCII).

5. For each segment perform the following substeps:
   
   (a) If the segment does not begin with the ACE prefix then leave the segment unchanged, otherwise save a copy of the segment and proceed to substep b.

   (b) Remove the ACE prefix.

   (c) Decode the segment using the decoding algorithm in [PUNYCODE] and catch any error. If there was an error then restore the saved copy from substep a.

6. Verify that at least one segment was altered in step 5.

7. Rejoin the segments into a single sequence, and save a copy of the result.

8. Apply ToASCII to the current sequence and to the saved copy from step 3.

9. Verify that the two results of step 8 match using a case-insensitive ASCII comparison.

10. Return the saved copy from step 7.

5. ACE prefix

[[ Note to the IESG and Internet Draft readers: The two uses of the string "iesg--" below are to be changed at time of publication to a prefix which fulfills the requirements in the first paragraph. IANA will assign this value. ]]

The ACE prefix, used in the conversion operations (section 4), is two ASCII letters followed by two hyphen-minuses. It cannot be the same as the prefix assigned to IDNA. The ToASCII and ToUnicode operations MUST recognize the ACE prefix in a case-insensitive manner.

[[ OPEN ISSUE: We might want to consider a prefix that uses different punctuation, or an infix that uses no punctuation. ]]

[[ OPEN ISSUE: We might want to consider using the same prefix as IDNA. ]]

The ACE prefix for IMAA is "iesg--" or any capitalization thereof.

This means that an ACE local part might be "foobar!iesg--de-jg4avbhy1noc0d!iesg--d9juau41awczczp", where "de-jg4avbhy1noc0d" and "d9juau41awczczp" are the parts of the ACE local part that are generated by the encoding steps in [PUNYCODE].

While every encoded segment (segment that would be altered by ToUnicode) within an ACE local part begins with the ACE prefix, not every segment beginning with the ACE prefix is an encoded segment. Segments that begin with the ACE prefix but are not encoded segments
will confuse users, and local parts containing such segments SHOULD NOT be used as mailbox names.

6. References

6.1 Normative references


6.2 Informative references


7. Security considerations

Because this document normatively refers to [IDNA], [NAMEPREP], [PUNYCODE], and [STRINGPREP], it includes the security considerations from those documents as well.

Internationalized local parts will cause mail addresses to become longer, and possibly make it harder to keep lines in a header under 78 characters. Lines that are longer than 78 characters (which is a SHOULD specification, not a MUST specification, in RFC 2822) could possibly cause mail user agents to fail in ways that affect security.

8. IANA considerations

IANA will assign the ACE prefix in consultation with the IESG, possibly following the same process used for [IDNA].

9. Authors’ addresses
Paul Hoffman
Internet Mail Consortium and VPN Consortium
127 Segre Place
Santa Cruz, CA 95060  USA
phoffman@imc.org

Adam M. Costello
University of California, Berkeley
http://www.nicemice.net/amc/