USENIX Association

Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

GNU Mailman, Internationalized

Barry A. Warsaw
Zope Corporation
barry@zope.com
barry@python.org
http://www.zope.com
http://barry.warsaw.us

Abstract

GNU Mailman is a mailing list management system that has been in production use since 1998. In December 2002, a version 2.1 was released containing many new features. This paper will describe one of the most important – Mailman 2.1's internationalization support. Presented here are the tools that were built and the approaches Mailman took to marking and translating text, as well a review of some of the benefits and pitfalls of Mailman's solution. Also presented will be some future directions for internationalized Mailman, as well as other complex Python applications such as Zope.

1 Introduction

GNU Mailman was invented by John Viega sometime before 1997, or so is indicated by the earliest known archived message on the subject. The earliest hit for "viega mailman" in Google groups is about the Dave Matthews band mailing list that John was running [Viega97].

At the time, python.org [Python.Org] was running a hacked version of Majordomo for all its special interest group (SIG) mailing lists, but this had two problems: first, the site was becoming unmaintainable as the administrators tried to customize new features into Majordomo, and second, it just wouldn't do to run Python's mailing lists on a Perl-based list server.

When Mailman was first released, python.org quickly adopted it and has been using it ever since. Mailman 2.0 marked a milestone in its development, as version 2.0.13 is quite stable, and deployed at thousands of sites. It runs everything from small special interest group lists to huge announcement lists, at sites ranging from the commercial (RedHat, SourceForge, Apple, Dell, SAP, and Zope Corporation), to the hacker community (XEmacs, Samba, Gnome, KDE, Exim, and of course Python), to numerous educational organizations and non-profits. There are lots of hosting facilities providing Mailman services, and increasingly, quite a few international organizations.

One of the reasons for the increased interest from the non-English speaking world is that Mailman 2.1, which had been in development for about two years, is fully internationalized. Internationalization is the process of preparing an application for use in multiple locales. Localization is the process of specializing the application for a specific locale. For example, during internationalization, all end-user displayable text in the Mailman 2.1 source code was specially marked as requiring translation. Mailman 2.1.1 (the latest available patch release at the time of writing), has been localized to almost 20 natural languages.

While Mailman 2.1 may appear to be only a minor revision over 2.0.13, it really represents quite an extensive rewrite. It could easily have been argued that this version should be called Mailman 3.0. Before describing the details of the internationalization work, a brief overview of Mailman is provided, including

a quick tour of some of the other important features in the 2.1 release.

2 What is GNU Mailman?

"GNU Mailman" (informally referred to as just "Mailman"), is a system for managing electronic mailing lists. It is implemented primarily in Python, an object-oriented, very high-level, open source programming language. Mailing lists are administered by a list owner, and users can interact with the list – including subscribing and unsubscribing – through the web and through email. Site administrators can also interact with Mailman via a suite of command line scripts, or even via the interactive Python prompt. Mailman is the official mailing list manager of the GNU project and is available under terms of the GNU General Public License [GPL].

Mailman strives for standards compliance, and as such is interoperable with a wide range of web servers and browsers, and mail servers and clients. Of the web servers, it requires the ability to execute CGI scripts, and of mail servers it requires the ability to filter messages through programs. Apache is probably the most widely used web server for Mailman, and any of the Big 4 mail servers (Sendmail, Postfix, Qmail, and Exim) will work just fine. The HTML that Mailman outputs is extremely pedestrian so just about any web browser should work with it, as long as it supports cookies. Mailman should work with any MIME-compliant mail reader. Mailman works on any Unix-like operating system, such as GNU/Linux.

Mailman supports a wide range of features, such as:

- User selectable delivery modes. Members can elect to receive messages immediately, or in batches called digests. Two forms of digests are supported, RFC 1153 style plain text digests [RFC1153], and MIME multipart/digest style digests. Non-digest deliveries can be personalized specifically for the recipient of the message. This means that various aspects of the message, i.e. the header or footer, or the `To` field, can contain information specific to the member receiving the message.

- Extensive privacy options which allow a list administrator to select policies for subscribing and unsubscribing (open, confirmation required, or approval required), policies for posting to the list (open, moderated, members only, approved posters only), and some limited spam defenses.

- Automatic bounce processing. Bouncing addresses are the bane of any mailing list, and Mailman provides two mechanisms for automatic bounce detection, regular expression based bounce matching and Variable Envelope Return Paths [VERP].

  RFC 3464 [RFC3464] and the older RFC it replaces [RFC1894] describe a standard format for bounce notifications. However, many mail systems ignore or incorrectly implement this standard. For recognizing bounce messages, Mailman has an extensive set of regular expression based matches used to dig the bouncing address out of the notice. For fool-proof bounce detection, Mailman also supports VERP, a technique where the intended recipient's address *as it appears on the mailing list* is encoded into the envelope sender of the message. Because remote mail servers are required to send bounces to the envelope sender, Mailman can unambiguously decode the intended recipient's address and register an accurate bounce. Note that technically, VERP must be implemented in the mail server, but Mailman's use of the technique is close enough to warrant the label.

- Archiving. Mailman comes bundled with an archiver called Pipermail. Pipermail's chief advantages are that it comes bundled, that it is implemented in Python, and that in Mailman 2.1 it is internationalized, allowing the display of messages in alternative languages and character encodings. Its primary disadvantages are that it doesn't support searching and isn't very customizable. Mail-

man is easily integrated with external archivers.

- A mail to news gateway. Mailman can be configured to gateway lists to and from Usenet newsgroups. For example, the comp.lang.python newsgroup is gatewayed to the python-list@python.org mailing list. Even moderated lists, such as comp.lang.python.announce can be gatewayed, with Mailman serving as the moderation tool.

- Auto-responder, content filtering, and topics. The auto-responder can be set up to send a canned message whenever someone posts to the list, or emails the list owner or -request robot. Content filtering allows the list owner to explicitly filter or pass specific MIME (Multipurpose Internet Mail Extensions [RFC2045]) content types. Topics allow the list owner to assign incoming messages to any of a configurable number of groups, and members can "subscribe" to a specific topic, receiving only the subset of list traffic that matches the desired topics.

- Virtual domains. Mailman can be used on a mail server that supports multiple virtual domains. For example, the python.org and zope.org mail domains are run on the same machine, from the same Mailman installation. The one limitation in Mailman 2.1 is that a mailing list with the same name may not appear in more than one domain. This restriction will be lifted in future versions.

Mailman also provides each list with its own home page (called a "listinfo" page) which can be customized through the web. Mailing lists can be automatically created and deleted through the web (with proper support from the mail server). Mailman also provides web-based approval of moderated messages and subscriptions. There are a host of other smaller new features in Mailman 2.1 which won't be described in this paper.

3 Internationalization Issues

The new features in Mailman 2.1 are extensive, but the most visible addition is the support for multiple natural languages. This means that all the administrative and publicly visible web pages, all the email notifications, and even the built-in archiver can be configured to produce text in any of nearly 20 natural languages out of the box. A large part of the re-architecting of Mailman for 2.1 has been to provide a framework for easily adding new natural languages as they become available from volunteer translation teams.

3.1 Message IDs

Not every string in an application needs to be translated. For example, some strings are used as keys in dictionaries, or represent mail headers, or contain HTML tags. To make the proper distinction we refer to strings that are intended for human readability as "text" or "messages". One of the most labor intensive parts of internationalizing an existing code base such as Mailman's is to go through every string in the software and distinguish messages from ordinary strings. In addition to the non-translatable strings described above, the decision was made to not translate log messages since these are not intended for the end-user, and would make debugging in global community more difficult.

Each message that is to be translated needs to have four pieces of information at runtime in order to calculate the translated text: the application domain, the message id, the default text, and the target locale. Because Mailman is a fairly self-contained application, there is only one static domain, the "mailman" domain, which never changes during the life of the program's execution.

The message id and default text are two related, but distinct concepts. The message id uniquely identifies the textual message to be displayed to the user. The message id names the message but it may not necessarily be the message. It is the message id which is the primary key into a translation catalog dictio-

nary.

The default text is the text to use as the translation of the message id, when the id is not found in the translation catalog. Because coordinating 20 different language teams is a project management challenge, it is common for some language catalogs to lag behind the source code development. Mailman releases are rarely delayed so that language teams can catch up (although advance notice of impending releases is usually given). It is often the case, therefore, that a particular message id won't be found in a specific language catalog. The default text is the fall back to use in this case.

As an example, suppose a web form had a Delete button. The message id for the button might be something like "form27-delete-button", while the default text might be "Delete".

Message ids may be explicit or implicit. In the above example "form27-delete-button" is an explicit message id. While it uniquely identifies the message to be used, it does not contain any text that will be displayed to the user. The advantage of explicit message ids is that they are immune to minor typos or formatting changes (e.g. whitespace or punctuation additions or deletions). The disadvantages of explicit message ids are two-fold: they require an extra catalog mapping message ids to the default language (e.g. English in Mailman's case), and they make the source code less readable. The latter is the more serious consequence; since nearly all human readable text in Mailman exists in Python source code, using explicit message ids would make the code nearly unreadable. A developer would have to consult the English catalog several times for some lines of code.

The alternative approach is to use implicit message ids, where the message id serves a dual purpose as the default text. Thus the human readable text that appears in the Python source code is first used as the message id, and if that fails to find a translation, it is used as the default text. While this has the advantage of making the source code more readable and easier to develop, it has several disadvantage. First, a message such as "Delete" which has one spelling in English, may be translated to one of several different words in another language, depending on the context. This poses a problem for the translator because the message id "Delete" may appear a dozen times in the application, but may require several different words in the target language. Also, minor changes in formating or punctuation change the message id, which requires a re-translation (this may be considered an advantage because changes in punctuation can cause semantic differences, requiring a re-translation anyway).

There is no perfect solution, but Mailman has decided to use implicit message ids because of the source code readability advantages. This occasionally requires negotiation between the application developers and the translation teams to choose appropriate and distinguishable message ids, and imposes a sort of inertia against changing existing text in the source code. One way to alleviate these problems in future releases would be to use a mix of implicit and explicit message ids, where implicit ids are used predominantly, but in rare cases explicit ids (along with a partial English catalog) are used to resolve ambiguities.

3.2 The Locale

Internationalizing a web-based application is much more complicated than internationalizing a command line program such as 'ls' because the natural language context (i.e. the "locale") is determined by the web request, or the email message being processed, instead of by the user's shell. In Mailman, the locale is dynamic and fluid; there may in fact be several locales needed to process any particular email message. Most of the existing techniques for internationalizing programs assume a static locale and a single domain. Mailman inherits the single domain tradition of these tools, but it uses dynamic techniques to calculate the translation locale.

We use the term "locale" and "language" interchangeably below, although this is not completely accurate. A locale describes much more than the language used; it also defines

the character encoding, as well as the formatting of dates, numbers, currency, etc. However, since Mailman does not currently support the localization of data such as dates, the language selection is the most important aspect of the active locale. Typically (although not exclusively) a single character encoding is used for a single language.

At any given point in the processing, the following locales may exist.

- The source code language. Mailman is developed in English, so by default, the English text is always available. Translations can be, and often are, incomplete and the English message is the global fall back.
- The site default language. Of the nearly 20 languages that Mailman supports out of the box, the site administrator can choose one of those languages as the "site default language". When no other locale is known, the site default will be used.
- The list default language. Every mailing list has a default language as well as as set of alternatively supported languages. The list default language is used for all the administrative pages. It is also the language used when the list context is known and there is no overriding context.
- The page default language. The list administrator can also choose to let the list support any other language allowed by the site administrator. A user browsing the list overview page can choose to view that page in any of those languages, by selecting that language in a pop up menu on the list's public web pages.
- The user's preferred language. The user can also choose one of the list supported languages to be their preferred language, by making this choice in their preferences page. All email notices that Mailman sends out to the user, or any web page that the user views when logged in, is displayed in the user's preferred language.

To support these multiple language contexts, Mailman uses an object-oriented approach where the locale is represented by an instance of a class. While this may seem natural, it is actually an elaboration of the global translation contexts in classic internationalized programs. This will be described in more detail later.

3.3 Character Encodings

Above and beyond the natural language issues, character encoding issues are probably the most vexing for the Mailman developers. "Character encoding" is usually referred to as the character set or charset, after the email header parameter described in RFC 2045.

A naive view would create a one-to-one correspondence between language and charset. For example, you might say that all Spanish text should be rendered in the iso-8859-1 (Latin-1) character set [ISOSoup]. However, even this simple example isn't accurate because the Euro sign is available only in iso-8859-15.

The problem is exacerbated by some Asian languages. Japanese for example may appear in any of euc-jp, iso-2022-jp, shift-jis, and may be different depending on whether the text appears in a web browser or in an email message. In fact, Mailman 2.1's naive approach causes some problems for Japanese users, especially when an email message is displayed as a web page in the archiver. This will be fixed in a future release.

Usually, English text uses the us-ascii character set, but for maximum interoperability, a list conducted in English may still want to be aware of Latin-1 characters. Mailman has to be careful when combining characters in different charsets, especially those for which us-ascii is not a subset.

For example, say a Spanish list received a message in Turkish, which uses Latin-5 (a.k.a. iso-8859-9). When that message is archived, different parts of the HTML page for the message will be in iso-8859-1 and other parts will be in iso-8859-9. But since HTML is inadequate at allowing multiple charsets in a single web page, the characters in one or the other

of those charsets must be converted to HTML entities, using their Unicode equivalent.

Multiple character set issues can also arise in the processing of email messages. Say for example that a message to a German list arrives in Japanese. Mailman has a feature called "headers and footers" which allow the list administrator to add some canned text to the start and end of a message (e.g. "To unsubscribe, click here"). Previous versions of Mailman would simply paste the header and/or footer around the original message body. This was broken for several reasons. The most obvious one is that if the message is really a Base64 encoded image, adding some spurious ASCII text around the original body would break the decoding. But if the message contained text in a different character set than the header or footer text, concatenation may render the original body unreadable. The solution requires careful examination of the original message, and in the extreme, ripping apart and reconstituting the structure of the original message, so that the headers and footers will always be added in a MIME-safe way.

Internationalization standards for email and HTML are defined in a series of RFCs, and these must be adhered to. For example, the most fundamental email RFC is 2822 [RFC2822] (which recently superseded RFC 822). This RFC describes the structure of an email message, but it is naive in its ASCII bias. RFCs 2045 through 2047 were added to address the use of multilingual character sets in email messages. RFC 2047 [RFC2047] was added to describe how non-ASCII characters are to be encoded in `Subject` fields and in other email headers. Mailman must be able to both interpret email messages with RFC 2047 encoded headers, and produce properly formatted ones when necessary. The challenge is to parse well intentioned, but erroneously encoded headers (to give the benefit of the doubt). These types of errors are all too common in email messages found in the wild and Mailman must be made robust against these types of poorly formed messages.

Prodded by these various issues, a comprehensive email package [Email] was developed and added to Python 2.2. The email package is compliant with all the relevant MIME RFCs, as well as other mail related standards.

3.4 Message Catalogs

GNU gettext [Gettext] is a widespread formal model for supporting multilingual applications in traditional C applications. Gettext encourages the use of implicit message ids. This leads to a rhythm whereby the C programmer marks translatable text in the source code by wrapping them in a function call. The function is usually _() – called "the underscore function" – and it has both a run-time behavior and an off-line purpose. At run-time, the underscore function performs the lookup of the message id in a global language catalog. There is also an off-line tool which searches all the source code for marked strings, extracting them and placing them in a message catalog template, called a .pot file.

GNU gettext contains both a C library and a suite of tools provided by The Translation Project [TranslationProject] to manage internationalized programs. The message extraction tool is called `xgettext`. While newer versions of `xgettext` understand Python source code to some degree, a pure-Python version of the program called `pygettext` was developed and is distributed with Python. `pygettext` has some additional benefit, including the ability to extract Python docstrings which may not be marked with the underscore function.

Mailman has adopted the gettext model of marking and translating source strings, and to that end, a GNU gettext-like standard module was implemented for Python [GettextModule]. While the gettext module implements the same global translation model of the C library, two elaborations were necessary for a more Pythonic interface.

First, for long running daemon processes such as Mailman 2.1's mail processor, multiple language contexts are required, so the global state implied by gettext isn't always appropriate. Here's an example to illustrate understand why.

When a new member subscribes to a mailing list, two notification messages can be sent. One is a welcome message sent to the member, and the other a new member notification sent to the list administrator. If the list's preferred language is Spanish, but the user prefers German, these two notifications will be sent out in two different languages. Since a single process crafts and sends both notifications, simply using _() wrapping doesn't give enough information. Which language should the underscore function translate its message id to?

Python solves this problem by providing an object-oriented API in additional to gettext's traditional functional API. Using the object interface, a program can create instances which represent the translation context; in other words, a single target language catalog is fully encapsulated in an object. For convenience, this object can be stored in some global context, and in the Mailman source, this global object can be saved and restored as necessary. Here is a simplified Python example:

```
# The list's preferred language is in
# effect right now
saved = i18n.get_translation()
try:
    i18n.set_language(
        users_preferred_language)
    send_user_notification()
finally:
    i18n.set_translation(saved)
send_admin_notification()
```

The second problem might be termed syntactic sugar or simple convenience, but it turns out to be extremely important in a Python program filled with translatable text. Python strings support variable substitution (also called "interpolation"), whereby a dictionary can be used to supply the substitutions. For example:

```
listname = get_listname()
member = get_username()
d = {'listname': listname,
     'member': member,
     }
print _('%(member)s has been '
        'subscribed to %(listname)s') % d
```

This is a critically important feature for internationalized programs because some languages may require a different order of the substitutions to be grammatically correct. While stock Python supports this requirement, its implementation leads to overly verbose code. In the above example, we've written the words "listname" and "member" four times each. Now imagine that level of verbosity duplicated a hundred times per source file. "Tedious" comes to mind!

Mailman solves this by providing its own underscore function, which wraps the gettext standard function, but provides a little bit of useful magic by looking up substitution variables in the local and global namespace of the caller. Using Mailman's special underscore function, the above code can then be rewritten as:

```
listname = get_listname()
member = get_username()
print _('%(member)s has been '
        'subscribed to %(listname)s')
```

While the average Perl programmer might ask what all the fuss is about, the Python programmer will notice something interesting: there's no interpolation dictionary and no modulus operator. The dictionary is created from the namespaces of the caller of the underscore function, which contains the "listname" and "member" local variables. The trick is that the underscore function uses a little known Python function called `sys._getframe()` to capture the global and local namespaces of the caller of underscore. It then puts these in an interpolation dictionary, with local variables overriding global variables, and then applies the modulo operator to the translated string, using this dictionary.

Marked translatable texts are used all over Mailman, and we run `pygettext` over all the source code to produce a gettext compatible mailman.pot catalog file. To translate this to a new language, the translation team would start by copying mailman.pot to messages/xx/LC_MESSAGES/mailman.po where "xx" is the language code for the new language. From here, standard tools such

as po-mode for Emacs or KDE's kbabel can be used to provide translations for all the source message ids. Then, standard gettext tools can be used to generate a mailman.mo binary file, which Python's gettext module can read. In this way, internationalized Python programs can leverage most of the tools translation teams normally use for C programs. Translators don't have to learn new tools just to translation Python programs.

3.5 Templates

While gettext style message text in Python source code are essential for an internationalized Mailman, they aren't always appropriate. For example, Mailman has always used templates as a way of conveniently representing full web pages or parts of email messages. These templates provide an easy way for site administrators to customize the look of the Mailman web pages, or the text sent out under various circumstances.

In an internationalized Mailman, the templates serve another purpose: they serve as a mechanism for providing language specific versions of the templates. Mailman uses almost 50 templates for various purposes, and of course provides the English versions of the templates as a default. Each supported language provides its own version of the templates, and Mailman has a defined search order for template lookup. For example, if Mailman were to display the public list overview page for a mailing list, it would search for the listinfo.html page, in the following locations (relative to the installation directory):

- The list-specific language directory lists/*listname*/*language*/*template*
- The virtual domain-specific language directory templates/*list.host_name*/*language*/*template*
- The site-wide language directory templates/site/*language*/*template*
- The global default language directory templates/*language*/*template*

The first location to yield the desired template wins. Thus, as with the gettext catalogs, English is always an available fall back.

Templates, like marked translatable source code text, support variable substitutions, using the same syntax. With templates, an explicit substitution dictionary is always provided, and the interpolation is performed after the template is located.

While the template system works well enough, its coarseness is a serious drawback. For example, say a new feature required the addition of an HTML button on one of the templates. While this is trivial to do for the English template, changing the English template means all the other templates are out-of-date. The translation teams must follow up with new versions of the modified templates, or other languages will lag behind the English version.

One solution for templates might be something like Zope Page Templates (ZPT) [Pelletier], and specifically, internationalized ZPT [I18NZPT]. Internationalized ZPT combines the best of gettext and templates by allowing the template author to design the template, marking sections of the template as translatable text. Another extraction tool can then run over the ZPT file and add the translatable messages to the overall catalog. This has the huge advantage that structural changes to a template don't require the translation teams to do any work. Changes to content messages in the template simply mean that one phrase may be out of date, but the whole template won't be invalidated.

4 Unicode

Python has two types of string objects, traditional 8-bit byte data strings and Unicode character strings. Python also has literal forms for each string type; quoted text are defined to be 8-bit strings unless the leading quote is prefixed with a "u", in which case it is a Unicode string. Because strings can come into Mailman in a variety of ways (e.g. through the web, an email message, or a

message catalog), the code must be prepared to handle encoded 8-bit strings and Unicode strings. Encoded 8-bit strings must be converted to Unicode via the `unicode()` built-in function in order to properly combine strings using concatenation or interpolation. In addition, Unicode strings must be re-encoded when printing them to certain streams, such as the log files, or standard output, but these encoding operations must watch out for unsupported characters. For example, if a Unicode string containing Latin-1 characters is printed to an ASCII-only terminal, a exception can be raised due to the non-ASCII characters in the string.

There is no doubt that character conversion issues have been the thorniest and most common bugs reported on Mailman 2.1 to date. While many issues have been fixed, the most important lesson learned is that Mailman should convert all text (not necessarily all strings!) to Unicode at the earliest possible time, ideally when the text enters the system. Mailman should use Unicode strings everywhere internally, converting to encoded 8-bit strings only where needed, and only at the last possible moment. Analysis will still be needed to decide how to handle conversion errors, such as those described above. In Python, the conversion function can be given an additional argument which specifies how strict the conversion should be, e.g. raise an exception if there are illegal characters found, throw the illegal characters away, or substitute a question mark for any illegal characters. The exact choice of the strictness flag will be dependent on the context in which the conversion is occurring.

5 Other Issues

There are some operational issues that need to be addressed for an internationalized application such as Mailman. Care must be taken when marking the source code for translation so that the text is split in a grammatically clear way. For example, whenever possible full sentences should be used, since translating sentence fragments may not be possible in all languages. Also, plural forms and genders pose particularly thorny problems. Python 2.3's gettext module supports plural forms, but only alpha releases of Python 2.3 have been made available as of this writing. English doesn't have gendered nouns, and sometimes, English text source strings need to be rewritten to accommodate translators.

Python supports a number of specific character encoding "codecs" in the standard distribution. While Python has built-in support for most Western codecs, Asian codecs in particular are not supported. Fortunately Japanese, Korean, and Chinese codecs are available as third party distributions.

Internationalization is a lot more than simply translating strings; many other values from currencies to dates must also be localized if they are to be displayed correctly for a particular language or country. Long term goals include wrapping IBM's ICU library [ICU] in Python.

While internationalization imposes some performance overhead, the effect is negligible. In an application such as Mailman, the performance of the mail server that Mailman feeds messages to, the network bandwidth, and the performance of the operating system and file system have a far greater influence on the performance of the system than does the Mailman software. Internationalization has imposed no perceived performance penalty.

Internationalization has increased the size of the software distribution, since by default the download contains the message catalogs for all supported languages. The current catalog contains over 1200 message ids and is approximately 228 KB in size. The translated and compiled catalog files are from 80 to 300 KB in size depending on the completeness of the translation. In all, the message catalogs themselves add approximately 16 MB to the uncompressed program source code. The templates add about another approximately 3 MB. For this reason, future releases of Mailman may provide an English-only distribution, with separately downloadable language packs.

6 Examples

Here is some sample Python code (reformatted for this paper) taken from Mailman 2.1 which shows marked messages:

```
label = _(categories[category])
realname = mlist.real_name
doc.SetTitle(
    _('%(realname)s Administration '
      '(%(label)s)'))
doc.AddItem(Center(Header(2, _(
    '%(realname)s mailing list '
    'administration<br>%(label)s '
    'Section'))))
```

Notice first that the local variables "label" and "realname" are referenced in the default text, and that their values come from the magic interpolation described above. In a translated message the order of the substitutions may change, so this ensures that the substitutions will occur in the grammatically correct location in the translated message. Also notice that there are actually three uses of the underscore function. In the second and third, the only function arguments are strings (in Python, adjacent strings are concatenated by the lexer). The `pygettext` rule for message extraction is a single string inside the underscore function, so both these texts will be extracted into the message catalog.

The first use of the underscore function is interesting in that it is getting a value out of a dictionary lookup. This is an example of a deferred translation, where the underscore function is only used for its run-time behavior. The text returned by the dictionary lookup is translated in a normal fashion, but the program source code `categories[category]` isn't extracted into the catalog because it isn't a string.

At the place where the categories dictionary is defined, the strings are also wrapped in an underscore function for `pygettext` extraction, but they aren't translated at that place in the program. We do this in a number of situations, such as when dictionaries are defined in module global scope. In that case, you would see something like:

```
def _(s): return s

categories = {
    'cat1': _('Privacy'),
    'cat2': _('Autoreplies'),
    'cat3': _('Topics'),
    }

_ = i18n._
```

Here, we're marking three strings for extraction, but we aren't translating them at the point of definition, because the target locale isn't known at this time.

Due to space limitations, sample web pages can't be included, however a public internationalized list can be viewed at `http://mail.python.org/mailman/listinfo/playground`. You can view this listinfo page in any of the languages supported by Mailman.

7 Future and Related Work

Internationalized Mailman servers are in deployed use around the world, and many of the earliest related bugs have been satisfactorily fixed. However the basic architecture used by Mailman may undergo additional refinement in future releases. In particular, Mailman will be rewritten to use Unicode internally for all human readable text. The templates used in Mailman will likely be redesigned to use something like Zope's ZPT, which allow finer grain control over the evolution of the templates.

The experiences learned during the Mailman internationalization effort have been carried forward to the Zope 3 internationalization effort [Zope3]. Zope is a web application server and framework, also written in Python. Zope's internationalization efforts are made more complicated by the fact that it is a framework supporting multiple applications rather than a single application. This means that while Mailman needs only a single application domain, Zope may have multiple simultaneous domains, even in a single translation context. Zope therefore needs a way to record the domain that a particular message

has come from so that it can be looked up in the proper catalog at output time. The solution has been to create a MessageID object, which is a subclass of string that contains the domain as an instance variable.

Also, in Zope few translatable messages are found in Python source code. The predominant carrier of human readable text is the ZPT. Thus the mechanisms described above for simple interpolation and global translation contexts aren't appropriate for Zope. While Zope's internationalization efforts are built on the Python tools developed during Mailman's internationalization, they will ultimately improve or expand on these tools.

As part of the Zope 3 internationalization effort, a Translation Web Service (TWS) has been proposed [ZopeTWS]. While largely only science fiction at the time of this writing, the TWS is a vision of how to coordinate the project management issues related to internationalization. One of the most difficult ongoing problems for an internationalized project is coordinating the output of the software developers with the translation efforts of the language teams. While the Translation Project attempts to automate and coordinate much of this process, the TWS plans to take this aspect a step further by providing a global web service for truly collaborative translations. With the TWS, a project manager could upload message templates for particular domains, and language teams would translate messages at their own pace. When a new version of the software is prepared for release, the project manager could then download a snapshot of the current state of the various translations for the project. The key advance with the TWS is that once the infrastructure is in place, the software developers are no longer bottlenecks in the translation effort, and coordination among translation team members is automatic.

8 Acknowledgments

The author would like to thank Zope Corporation (`http://www.zope.com`) for their support of this work.

Mailman was originally invented by John Viega, and at various times has been shepherded, maintained, and developed by Thomas Wouters, Ken Manheimer, Harald Meland, and Scott Cotton. The author is the current project leader.

Juan Carlos Rey Anaya and Victoriano Giralt produced the first working prototypes of an internationalized Mailman, and worked with the author to designed the architecture for supporting internationalization. Others who provided invaluable contributions for the internationalization effort include Ben Gertzfield, Martin von Loewis, Simone Piunno, Daniel Buchmann, Tokio Kikuchi, and Ousmane Wilane. The ACKNOWLEDGEMENTS file that comes with the Mailman source distribution contains a detailed list of contributors.

9 Availability

GNU Mailman is free software, covered by the GNU General Public License. It is available for download from `http://sf.net/projects/mailman`

More information on Mailman can be found at its home page `http://www.list.org`

Mirrors of the Mailman site are at `http://www.gnu.org/software/mailman` and `http://mailman.sf.net`

References

[Viega97] `http://groups.google.com/groups?q=viega+mailman&hl=en&lr=&ie=UTF-8&scoring=d&start=60&sa=N&filter=0`

[Python.Org] *Python Home Page*, `http://www.python.org`

[GPL] Free Software Foundation, *GNU General Public License*, `http://www.gnu.org/licenses/gpl.txt`

[RFC1153] F. Wancho, *Digest Message Format*, http://www.faqs.org/rfcs/rfc1153.html

[VERP] D. J. Bernstein, *Variable Envelope Return Paths*, http://cr.yp.to/proto/verp.txt

[RFC3464] K. Moore and G. Vaudreuil, *An Extensible Message Format for Delivery Status Notifications*, http://www.faqs.org/rfcs/rfc3464.html

[RFC1894] K. Moore and G. Vaudreuil, *An Extensible Message Format for Delivery Status Notifications*, http://www.faqs.org/rfcs/rfc1894.html

[RFC2045] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, http://www.faqs.org/rfcs/rfc2045.html

[ISOSoup] *The ISO 8859 Alphabet Soup*, http://czyborra.com/charsets/iso8859.html

[Gettext] *GNU gettext*, http://www.gnu.org/software/gettext/gettext.html

[TranslationProject] *The Translation Project Site*, http://www.iro.umontreal.ca/contrib/po/HTML/index.html

[GettextModule] *Multilingual internationalization services*, http://www.python.org/doc/current/lib/module-gettext.html

[RFC2822] P. Resnick, *Internet Message Format*, http://www.faqs.org/rfcs/rfc2822.html

[RFC2047] K. Moore, *MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text*, http://www.faqs.org/rfcs/rfc2047.html

[Email] *email – an email and MIME handling package*, http://www.python.org/doc/current/lib/module-email.html

[Pelletier] M. Pelletier and A. Latteier, *The Zope book*, Chapter 5, Using Zope Page Templates, ISBN 0735711372.

[I18NZPT] http://dev.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/ZPTInternationalizationSupport

[ICU] *International Components for Unicode*, http://www-124.ibm.com/icu/

[Zope3] *Welcome to the Zope 3 project*, http://dev.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/

[ZopeTWS] *Translation Web Service*, http://dev.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/TranslationWebService