Internationalization

Contents

This design explains how the Apertis platform will be made localizable and how it will be localized to specific locales.

"Internationalization"("i18n") is the term used for the process of ensuring that a software component can be localized. "Localization"("l10n") is the process of adding the necessary data and configuration so an internationalized software adapts to a specific locale. A locale is the definition of the subset of a user's environment that depends on language and cultural conventions.

All this will be done with the same tools used by GNOME and we do not anticipate any new development in the middleware itself, though UI components in the Apertis shell and applications will have to be developed with internationalization in mind, as explained in this document.

For more detailed information of how translation is done in the FOSS world, a good book on the subject is available[1].

Internationalization

Text input

Some writing systems will require special software support for entering text, the component that provides this support for an specific writing system is called input method. There is a framework for input methods called IBus[2] that is the most common way of providing input methods for the different writing systems. Several input methods based on IBus are available in Ubuntu, and it is very unlikely that any needs will not be covered by them. An older, but more broadly-supported, input method framework is SCIM[3] and an even older one is

[1] http://archive.flossmanuals.net/open-translation-tools/
[2] http://en.wikipedia.org/wiki/Intelligent_Input_Bus
[3] http://en.wikipedia.org/wiki/Smart_Common_Input_Method

XIM[4].

The advantage of using an input method framework (instead of adding the functionality directly to applications or widget libraries) is that the input method will be usable in all the toolkits that have support for that input method framework.

Note that currently there is almost no support in Clutter for using input methods. Lead Clutter developer Emmanuele Bassi recommends doing something similar to GNOME Shell, which uses `GtkIMContext`[5] on top of `ClutterText`[6], which would imply depending on GTK+. There's a project called clutter-imcontext that provides a simple version of GtkIMContext for use in Clutter applications, but Emmanuele strongly discourages its use. GTK+ and Qt support XIM, SCIM and IBus.

In order to add support for GtkIMContext to ClutterText, please see how it's done in GNOME Shell[7]. As can be seen this implementation calls the following functions from the GtkIMContext[8] API:

- `gtk_im_context_set_cursor_location`
- `gtk_im_context_reset`
- `gtk_im_context_set_client_window`
- `gtk_im_context_filter_keypress`
- `gtk_im_context_focus_in`
- `gtk_im_context_focus_out`

Between the code linked above and the GTK+ API reference it should be reasonably clear how to add GtkIMContext support to Clutter applications, but there's also the possibility of reusing that code instead of having to rewrite it. In that case, we advise to take into account the license of the file in question (LGPL v2.1).

For systems without a physical keyboard, text can be entered via a virtual keyboard. The UI toolkit will invoke the on-screen keyboard when editing starts, and will receive the entered text once it has finished. So the on-screen keyboard can be used for text input by a wide variety of UI toolkits, Collabora recommends it to use IBus.

The reasons for recommending to use an input-method framework is that most toolkits have support for it, so if an application is reused that uses Qt, the on-screen keyboard will be used without any specific modification, which wouldn't be the case if `GtkIMContext` would be used.

[4] http://www.x.org/releases/X11R7.6/doc/libX11/specs/XIM/xim.html

[5] http://developer.gnome.org/gtk/unstable/GtkIMContext.html#GtkIMContext.description

[6] https://developer.gnome.org/st/stable/StEntry.html

[7] http://git.gnome.org/browse/gnome-shell/tree/src/st/st-im-text.c

[8] http://developer.gnome.org/gtk/unstable/GtkIMContext.html#GtkIMContext.description

About why to use IBus over other input-method frameworks, the reason is that IBus is already supported by most modern toolkits, has a very active upstream community and the cost of developing input-methods with IBus is lower than with other frameworks. Currently, IBus is the default input method framework in Ubuntu and Fedora, and GNOME is considering dropping support for other frameworks'input methods.

Text display

For text layout and rendering the toolkit needs to support all writing systems we are interested in. GTK+ and Clutter use Pango which supports a very broad set of natural language scripts. The appropriate fonts need to be present so Pango can render text.

The recommended mechanism for translating those pieces of text that are displayed in the UI is to export those strings to a file, get them translated in additional files and then have the application use at runtime the appropriate translated strings depending on the current locale. GNU gettext implements this scheme and is very common in the FOSS world. Gettext also allows adding a comment to the string to be translated, so it gives more context that can aid the translator to understand better how the string is used in the UI. This additional context can also be used to encode additional information as explained later. The GNU gettext[9] manual is comprehensive and covers all this in detail.

This is an example of all the metadata that a translated string can have attached:

```
#. Make sure you use the IEC equivalent for your language
## Have never seen KiB used in our language, so we'll use KB
#: ../glib/gfileutils.c:2007
#, fuzzy, c-format

msgctxt "File properties dialog"
msgid "%.1f KiB"
msgstr "%.1f KB"
```

For strings embedded inside [ClutterScript] files, ClutterScript supports a `translatable` property to mark the string as translatable. So to mark the text of a `ClutterText` as translatable, the following ClutterScript should be used:

```
"label" : {
    "text" : {
        "translatable" : true,
        "string" : "Label Text"
    }
}
```

[9]http://www.gnu.org/software/gettext/manual/gettext.html

Note that `clutter_script_set_translation_domain()` or `textdomain()`[10] needs to be called before translatable strings can be used in a ClutterScript file.

gettext[11] currently does not support extracting strings from ClutterScript files; support for that needs to be added.

Previous versions of this document recommended using intltool[12]. However, in recent years, it has been superceded by gettext[13]. Previously, gettext was unmaintained, and intltool was developed to augment it; now that gettext is actively maintained and gaining new features, intltool is no longer necessary.

Message IDs It is most common in FOSS projects (specially those using GNU gettext) to use the English translation as the identifier for the occurrence of a piece of text that needs to be translated, though some projects use an identifier that can be numeric (`T54237`) or a mnemonic (`PARK_ASSIST_1`). The IDs will not leak to the UI if the translations are complete, and there is also the possibility of defining a fallback language.

There's two main arguments used in favor of using something other than plain English as the ID:

- so that when the English translation is changed in a trivial way, that message isn't marked as needing review for all other languages;
- and to avoid ambiguities, as "Stop"may refer to an action or a state and thus may be translated differently in some languages, while using the IDs `state_stop` and `action_stop` would remove that ambiguity.

When using gettext, the first argument loses some strength as it includes a tool that is able to merge the new translatable string with the existing translations, but marking them as in need of review. About the argument of avoiding ambiguity, GNU gettext was extended to provide a way of attaching additional context to a message so that is not a problem anymore.

Regarding advantages of using plain English (or other natural language) as the message ID:

- better readability of the code,
- when the developers add new messages to the application and run it, they will see the English strings which is closer to what the user will see than any other kind of IDs.

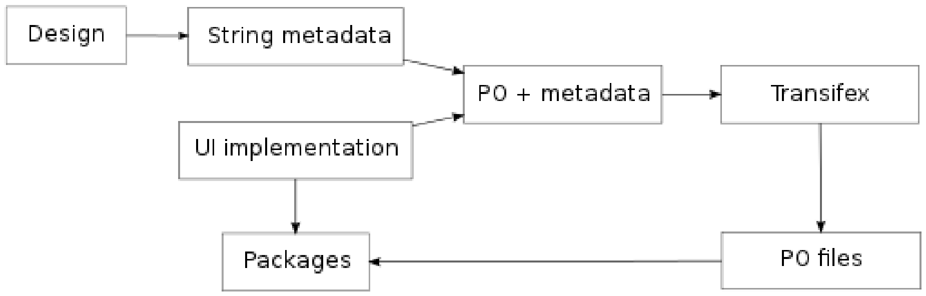
From the above it can be understood why it's normally recommended to just use the English translation as the placeholder in the source code when using GNU gettext.

[10] https://manpages.debian.org/unstable/gettext-base/textdomain.3.en.html
[11] http://www.gnu.org/software/gettext/manual/gettext.html
[12] https://launchpad.net/intltool/
[13] http://www.gnu.org/software/gettext/manual/gettext.html

Regarding consistency, there's a slight advantage in using natural language strings because when entering translations the translation software may offer suggestions from the translation memory and given that the mnemonic IDs are likely to be unique, there will be less exact matches.

Because of the need to associate to each translation metadata such as the font size and the available space, plus having product variants that share most of the code but can have differences in fonts and widget sizes, we recommend to use mnemonics as IDs, which would allow us to keep a list of the translatable strings and their associated fonts and pixels for each variant. This will be further discussed in Testing.

This diagram illustrates the workflow that would be followed during localization.



For better readability of the source code we recommend that the IDs chosen suggest the meaning of the string, such as *PARK_ASSIST_1*. Instead of having to specify whole font descriptions for each string to translate, Collabora recommends to use styles that expand to specific font descriptions.

Here is an example of such a metadata file, note the font styles NORMAL, TITLE and APPLICATION_LIST:

```
PARK_ASSIST_1 NORMAL 120px
PARK_ASSIST_2 NORMAL 210px
SETTINGS_1 TITLE 445px
BROWSER APPLICATION_LIST 120px
```

And here is the PO file that would result after merging the metadata in, ready to be uploaded to Transifex:

```
#. NORMAL,120px
#: ../preferences.c:102
msgid "PARK_ASSIST_1"
msgstr "Park assist"
#. NORMAL,210px
#: ../preferences.c:104
msgid "PARK_ASSIST_2"
msgstr "Park assist"
```

If for some reason some source code is reused that uses English for its translation

IDs and the rest of the application or library uses synthetic IDs, Collabora recommends to have a separate domain for each section of the code, so all English IDs are in their own PO file and the synthetic IDs in their own. In this case, note that matching metadata to individual strings can be problematic if the metadata isn't updated when the string IDs change. It will be a problem as well if there are several occurrences of exactly the same string.

When it is needed to modify the metadata related to existing strings, the process consists of modifying the file containing string metadata, then merging it again with the PO files from the source code and importing it into the translation management system.

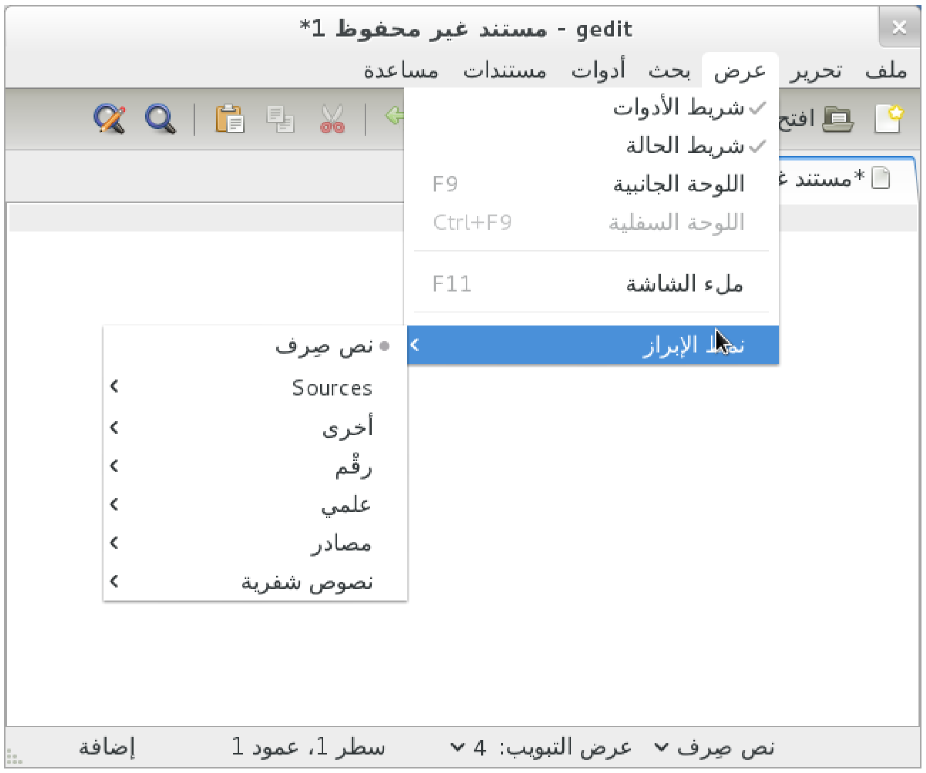
Consistency Translation management systems offer tools to increase the consistency of the translations, so the same words are used to explain the same concept. One of the tools that Transifex offers is a search feature that allows to quickly check how a word has been translated in other instances. Another is the *translation memory* feature, which suggests translations based on what has been translated already.

There isn't any relevant difference in how these tools work and whether the strings are identified by synthetic IDs or by their English translations.

UI layout

Some languages are written in orientations other than left to right and users will expect that the UI layout takes this into account. This means that some horizontal containers will have to layout its children in reverse order, labels linked to a widget will also be mirrored, and some images used in icons will have to be mirrored horizontally as well.

Here is an example of an application running under a locale whose orientation is right-to-left, note the alignment of icons in the toolbar and the position of the arrows in submenus:



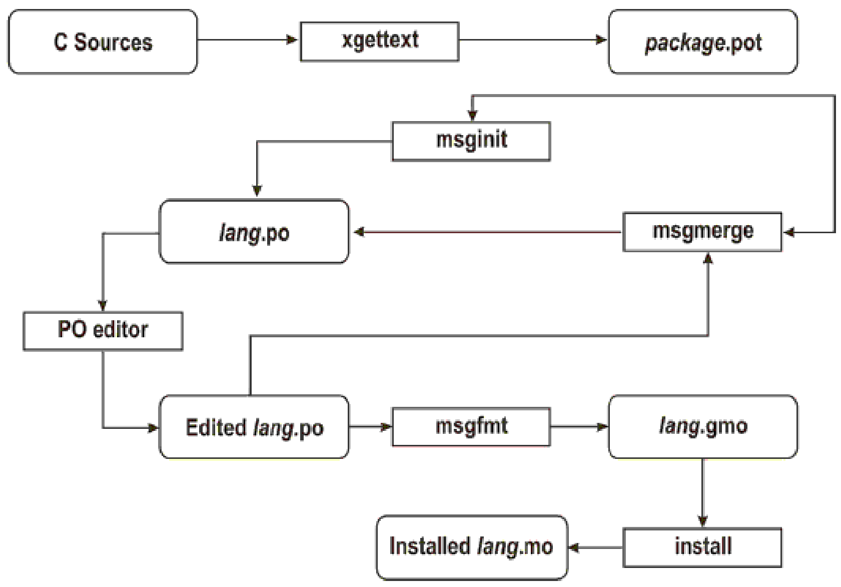
Localization

Translation

GNU gettext Most of the work happens in the translation phase, in which .po files are edited so they contain appropriate translations for each string in the project. As illustrated in the diagram below, the .po files generated from the original .pot file serve as the basis for starting the translation. When the source code changes and thus a different .pot file gets generated, GNU gettext includes a tool for merging the new .pot file into the existing .po files so translators can work on the latest code.

This diagram illustrates the workflow[14] when using GNU gettext to translate text in an application written in C:

[14] http://upload.wikimedia.org/wikipedia/commons/0/05/GNU_gettext_process.png



GNU gettext Working Process

From time to time, it is needed to extract new translatable strings from the source code and update the files that are used by translators. The extraction itself is performed by the tool xgettext[15], which generates a new POT file containing all the translatable strings plus their locations in the source code and any additional context.

These are the programming languages[16] supported by GNU gettext: C, C++, ObjectiveC, PO, Python, Lisp, EmacsLisp, librep, Scheme, Smalltalk, Java, JavaProperties, C#, awk, YCP, Tcl, Perl, PHP, GCC-source, NXStringTable, RST and Glade.

The POT file and each PO file are fed to msgmerge[17] which merges the existing translations for that language into the POT file. Strings that haven't been changed in the source code get automatically merged and the remaining are passed through a fuzzy algorithm that tries to find the corresponding translatable string. Those strings that had a fuzzy match are marked as needing review. If strings are indexed with unique IDs instead of the English translation, then it's recommended to use the –no-fuzzy-matching option to msgmerge, so new

[15]http://www.gnu.org/savannah-checkouts/gnu/gettext/manual/html_node/xgettext-Invocation.html

[16]http://www.gnu.org/savannah-checkouts/gnu/gettext/manual/html_node/xgettext-Invocation.html#index-supported-languages_002c-_0040cod

[17]http://www.gnu.org/savannah-checkouts/gnu/gettext/manual/html_node/msgmerge-Invocation.html

IDs will be always empty. Otherwise, if the POT file contained already an entry for `PARK_ASSIST_1` and `PARK_ASSIST_2` was added, when merging into existing translations, the existing translation would be reused, but marking the entry as fuzzy (which would cause Transifex to use that translation as a suggestion).

Translation management Though these file generation steps can be executed manually with command line tools and translators can work directly on the `.po` files with any text editor, there are more high-level tools that aim to manage the whole translation process. Next we briefly mention the ones most commonly used in FOSS projects.

Pootle[18], Transifex[19] and Launchpad Rosetta[20] are tools which provide convenient UIs for translating strings. They also streamline the process of translating strings from new `.pot` versions and offer ways to transfer the resulting .po files to source code repositories.

Pootle is the oldest web-based translation management system and is mature but a bit lacking in features. Maintaining an instance requires a fair amount of experience.

Transifex is newer and was created to accommodate better than Pootle to the actual workflows of most projects today. Its UI is richer in features that facilitate translation and, more importantly, has good commercial support (by Indifex). It provides as well an API that can be used to integrate it with other systems. See Transifex for more details.

Launchpad is not easily deployable outside launchpad.net and is very oriented to Ubuntu's workflow, so we do not recommend its usage.

Both Pootle and Transifex have support for translation memory, which aids in keeping the translation consistent by suggesting new translations based on older ones.

If for some reason translators prefer to use a spreadsheet instead of web UIs or manually editing the PO files, csv2po[21] will convert a PO file to a spreadsheet and will convert it back so the translation system can be refreshed with the new translations.

po2csv will convert a PO file to a CSV one which has a column for the comments and context, another for the *msgid* and one more for the translation for the given language. *csv2po* will do the opposite conversion.

It's very likely that the CSV format that these tools generate and expect doesn' t match exactly what it is needed, so an additional step will be needed that

[18]http://pootle.translatehouse.org/?id=pootle/index
[19]https://www.transifex.com/
[20]https://translations.launchpad.net/
[21]http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/csv2po.html?id=toolkit/csv2po

converts the CSV file to the spreadsheet format required, and a step that does the opposite.

Transifex In this section we discuss in more details some aspects of Transifex. For an overview on other features of Transifex, please see the documentation for management[22] and translation[23].

Deployment options Transifex is available as a hosted web service in *http://www.transifex.com*[24] and there are

several pricing options[25] depending on the project size, features and level of technical support desired.

The FOSS part of Transifex is available as Transifex Community Edition and can be freely downloaded and installed in any machine with a minimally modern and complete Python installation. This version lacks some of the features that are available in *http://transifex.com*[26] and in the Enterprise Edition. The installation manual for the community edition is in `http://help.transifex.net/server/install.html`.

The hosted and the enterprise editions support these features in addition of what the community edition supports:

- Translation memory
- Glossary
- Improved collaboration between translators
- Improved UI theme

The advantage of the hosted edition is that it is updated more frequently (weekly) and that in the future it will be possible to order paid translations through the platform.

Transifex currently cannot estimate the space that a given translation will take and will need to be extended in this regard.

It also fully supports using synthetic translation IDs instead of English or other natural language.

Finally, Indifex provides commercial support for the enterprise edition of Transifex, which can either be self-hosted or provided as SaaS. Their portfolio includes assistance with deployment, consultancy services on workflow and customization, and a broad package of technical support.

[22]https://www.transifex.com/features/team-management/
[23]https://www.transifex.com/features/translation-tools/
[24]http://www.transifex.com/
[25]https://www.transifex.com/pricing/
[26]http://transifex.com/

Maintenance Most maintenance is performed through the web interface, by registered users of the web service with the appropriate level of access. This includes setting up users, teams, languages and projects. Less frequent tasks such as instance configuration, software updates, performance tuning and set up of automatic jobs are performed by the administrator of the server hosting the service.

Translation memory Transifex will provide suggestions when translating a string based on existing translations[27] in the current module or in other modules that were configured to share their translation memory[28]. This memory can also be used to pre-populate translations for a new module based on other modules' translations[29].

Glossary Each project has a series of terms that are very important to translate consistently or that can have several different possible translations with slightly different meanings. To help with this, Transifex provides a glossary[30] that will assist translators in these cases.

POT merging As explained in GNU Gettext, new translatable strings are extracted from the source files with the tool xgettext and the resulting POT file is merged into each PO file with the tool msgmerge.

Once the PO files have been updated, the tool tx (command-line transifex client) can be used to submit the changes to the server, this merge happening as follows[31]:

Here's how differences between the old and new source files will be handled:

- New strings will be added.
- Modified strings will be considered new ones and added as well.
- Strings which do not exist in the new source file (including ones which have been modified) will be removed from the database, along with their translations.

Keep in mind, however, that old translations are kept in the Translation Memory of your project.

Note that this process can be automated.

Automatic length check Transifex's database model will have to be updated to store additional metadata about each string such as the font description and the available size in pixels. The web application could then check how many

[27] https://www.transifex.com/features/translation-tools/
[28] https://docs.transifex.com/translation-memory/sharing-tm
[29] https://docs.transifex.com/translation-memory/enabling-autofill/
[30] https://docs.transifex.com/glossary/glossary
[31] https://docs.transifex.com/client/push

pixels the entered string would take in the UI, using Pango and Fontconfig[32]. For better accuracy, the exact fonts that will be used in the UI should be used for this computation.

Alternatively, there could be a extra step after each translation phase that would spot all the strings that may overflow and mark them as needing review.

Testing

Translations will be generally proof-read, but even then we recommend testing the translations by running the application to catch a number of errors which are noticeable only at run time. This run-time evaluation can spot confusing or ambiguous wording, as well as layout problems.

Each translation of a single piece of text can potentially require a wildly-differing width due to varying word and expression sizes in different languages. There are ways for the UI to adapt to the different string sizes but there are limits to how well this can work, so translators need often to manually check whether their translation fits nicely in the UI.

One way to automatically avoid many instances of layout errors would be to have available, during translation and along with the extracted strings, the available space in pixels and the exact font description used to display the string. This information would allow automatic calculation of string sizes, thus being able to catch translations that would overflow the boundaries. As explained in Message IDs, this metadata would be stored in a file indexed by translation ID and would be merged before importing it into the translation management software, which could use it to warn when a translated string may be too long. For this to consistently work, the translation IDs need to be unique (and thus synthetic).

When calculating the length of a translation for a string that contains one or more printf placeholders[33], the width that the string can require when displayed in the UI grows very quickly. For example, for the placeholder `%d` which can display a 32-bit integer value, the final string can take up to 10 additional digits. The only way to be safe is to assume that each placeholder can be expanded to its maximum size, though in the case of strings (placeholder `%s`) that is practically unlimited.

If, despite automatically warning the translator when a translation will not fit in the UI, some strings are too long, the UI widget that displays the string could ellipsize it to indicate that the displayed text isn't complete. If this occurred in a debug build, a run-time warning could be also emitted. These warnings would be logged only once a translated string has been displayed in the UI and wouldn't apply to text coming from an external input.

For manual testing, an image could be provided to translators so they could easily merge their work and test the software in their locale.

[32] http://fontconfig.org/

[33] http://pubs.opengroup.org/onlinepubs/9699919799/functions/printf.html

Other locale configuration

There is some other configuration that is specific to a locale but that is not specific to the application. This includes number, date and time formats, currency and collation. Most locales are already present in GNU glibc so we would only have to add a locale if it would target an extremely small population group.

Distribution

There are three main ways of packaging translations:

- package all the MO files (compiled PO files) along the rest of the files for a single component (for example gnome-shell in Ubuntu).
- package the MO files for a single component (usually a big one such as LibreOffice or KDE) and a specific language in a separate package (for example, firefox-locale-de[34] in Ubuntu).
- package several MO files corresponding to several components for one language (for example language-pack-cs-base in Ubuntu).

Our recommendation at this stage is to have:

- each application along with all its existing translations in a single package. This way the user will install e.g. `navigation-helper_1.10_armhf.deb` and the user will be able to switch between all the supported languages without having to install any additional packages.
- the rest of the MO files (those belonging to the UI that is pre-installed, such as applications and the shell) would be packaged grouped by language, e.g. `apertis-core-de_2.15_armhf.deb`. That way we can choose which languages will be pre-installed and can allow the user to install additional languages on demand.

If we do not want to pre-install all the required fonts and input methods for all supported languages, we could have meta-packages that, once installed, provide everything that is required to support a specific language. The meta-package in Ubuntu that provides support for Japanese is a good example of this[35].

Note that our current understanding is that the whole UI will be written, not reusing any existing UI components that may be present in the images. This implies that though some middleware components may install translations, those are not expected to be seen by the user ever.

This table should help make an idea of the sizes taken by packages related to localization:

Package name	Contents	Package size	Installed size
language-pack-de-base	MO files for core packages1	2,497 kB	8,432 kB

[34] http://packages.ubuntu.com/oneiric/firefox-locale-de
[35] http://packages.ubuntu.com/hardy/language-support-ja

Package name	Contents	Package size	Installed size
firefox-locale-de	German translation for Firefox2	233 kB	453 kB
libreoffice-l10n-de	Resource files with translations, and templates3	1,498 kB	3,959 kB
language-support-fonts-ja	Fonts for rendering Japanese	29,006 kB	41,728 kB
Ibus-anthy	Japanese input method4	388 kB	1,496 kB

The *language-support-fonts-ja* package is a virtual one that brings the following other packages (making up the total of 41,728 kB when installed):

Package name	Contents	Package size	Installed size
ttf-takao-gothic	Japanese TrueType font set, Takao Gothic Fonts	8,194.6 kB	12,076.0 kB
ttf-takao-pgothic	Japanese TrueType font set, Takao P Gothic Font	4,195.4 kB	6,196.0 kB
ttf-takao-mincho	Japanese TrueType font set, Takao Mincho Fonts	16,617.9 kB	23,456.0 kB

Modern distributions will bring all those fonts for Japanese-enabled installations, but depending on the commercial requirements, a system could make with just a subset. Similarly, other locales will require a set of fonts for properly rendering text in the same way as users in specific markets expect. In order to recommend specific font files, knowledge on the requirements are needed.

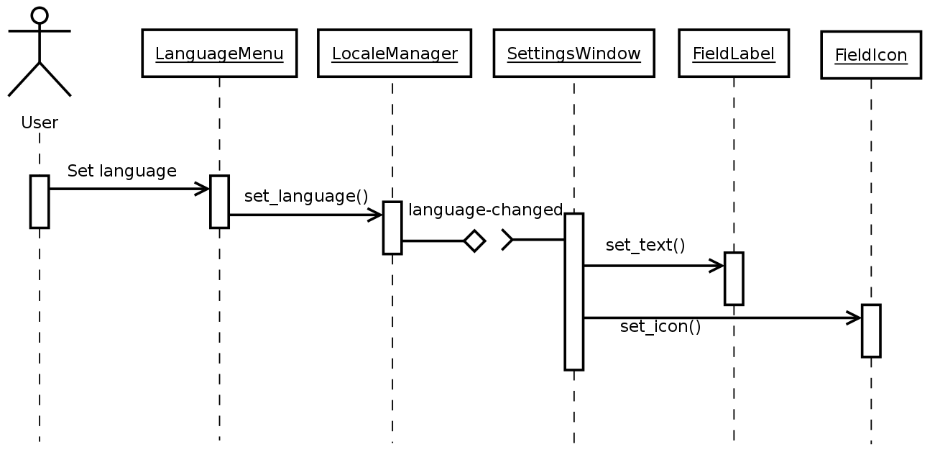
Runtime switching of locale

Common pattern

A usual way of implementing switching languages during runtime is to have those UI components that depend on the language to listen for a signal that gets emitted by a global singleton when the language changes. Those components will check the new language and update strings and probably change layout if the text direction has changed. Some other changes may be needed such as changing the icons, colors, etc.

The Qt toolkit has a bit of support for this solution and their documentation[36] explains in detail how to implement it. This can be easily implemented in Clutter and performance should be good provided that there isn't an excessive amount of actors in the stage.

[36]https://wiki.qt.io/How_to_create_a_multi_language_application



`LocaleManager` in the diagram would be a singleton that stores the current locale and notifies interested parties when it changes. The current locale would be changed by UI elements such as a combo-box in the settings panel, a menu option, etc.

Other UI elements that take locale-dependent decisions (in the diagram, `SettingsWindow`) would register to be notified when the locale changes, so they can change their UI (update strings, change icons, change text orientation, etc.).

Since systemd version 30, the systemd-localed service[37] has been provided as a standard D-Bus API (`org.freedesktop.locale1`) for managing the system' s locale, including being notified when it is changed, getting its current value, and setting a new value. This should be used in combination with the `org.gnome.system.locale` GSettings schema, which stores the *user's* locale preferences. We suggest that the `LocaleManager` from the diagram is implemented to query `org.gnome.system.locale` and returns the value of its `region` setting if set. If not set, the user is using the default system locale, which `LocaleManager` should query from `org.freedesktop.locale1`.

`org.freedesktop.locale1` is provided as a D-Bus API only, and `org.gnome.system.locale` is a GSettings schema. They are accessed differently, so a set of wrapper functions should be written as a convenience for application developers.

systemd-localed uses polkit[38] to authorise changes to the system locale, so vendors would need to write a policy which determines which applications are permitted to change the system locale, and which are allowed to query it. The default should be that only the system preferences application is allowed to change the locale; and all applications are allowed to query it (and be notified of changes to the locale).

[37] https://www.freedesktop.org/wiki/Software/systemd/localed/
[38] https://www.freedesktop.org/wiki/Software/polkit/

These snippets show how systemd-localed could be used by an application (omitting asynchronous calls for simplicity):

The following example shows how the user's locale can be queried by an application, first checking `org.gnome.system.locale`, then falling back to `org.freedesktop.locale1` if the user is using the system locale. It is expected that most of the code in this example would be implemented in the `LocaleManager`, rather than being reimplemented in every application.

{{ ../examples/locale-region-changed.c }}

Application helper API

To reduce the amount of work that most application authors will have when making their applications aware of runtime locale switches, we recommend that the SDK API includes a subclass of `ClutterText` (let's call it `ExampleText`) that reacts to locale changes.

`ExampleText` would accept a translatable ID via the function `example_text_set_text()`, would display its translation based on the current locale and would also listen for locale changes and update itself accordingly.

So xgettext can extract the string IDs that get passed to `ExampleText`, it would have to be invoked with `--flag=example_text_set_text:1:c-format`.

If applications use `ExampleText` instead of `ClutterText` for the display of all their translatable text, they will have to interface with `LocaleManager` only if they have to localize other aspects such as icons or container orientation.

Localization in GNOME

GNOME uses a web application called Damned Lies to manage their translation work-flow and produce statistics to monitor the translation progress. Damned Lies is specifically intended to be used within GNOME, and its maintainers recommend other parties to look into a more generic alternative such as Transifex. There used to be a separate tool called Vertimus but it has been merged into Damned Lies.

Participants in the translation of GNOME belong to translation teams, one for each language to which GNOME is translated, and they can have one of three roles: translator, reviewer and committer. As explained in GNOME's wiki[39]:

> *Translators contains persons helping with GNOME translations into a specific language, who added themselves to the translation team. Translators could add comment to a specific PO file translation, could reserve it for translations and could suggest new translations by upload a new PO file. The suggested translations will be reviewed by other team members.*

[39]https://wiki.gnome.org/TranslationProject/ContributeTranslations

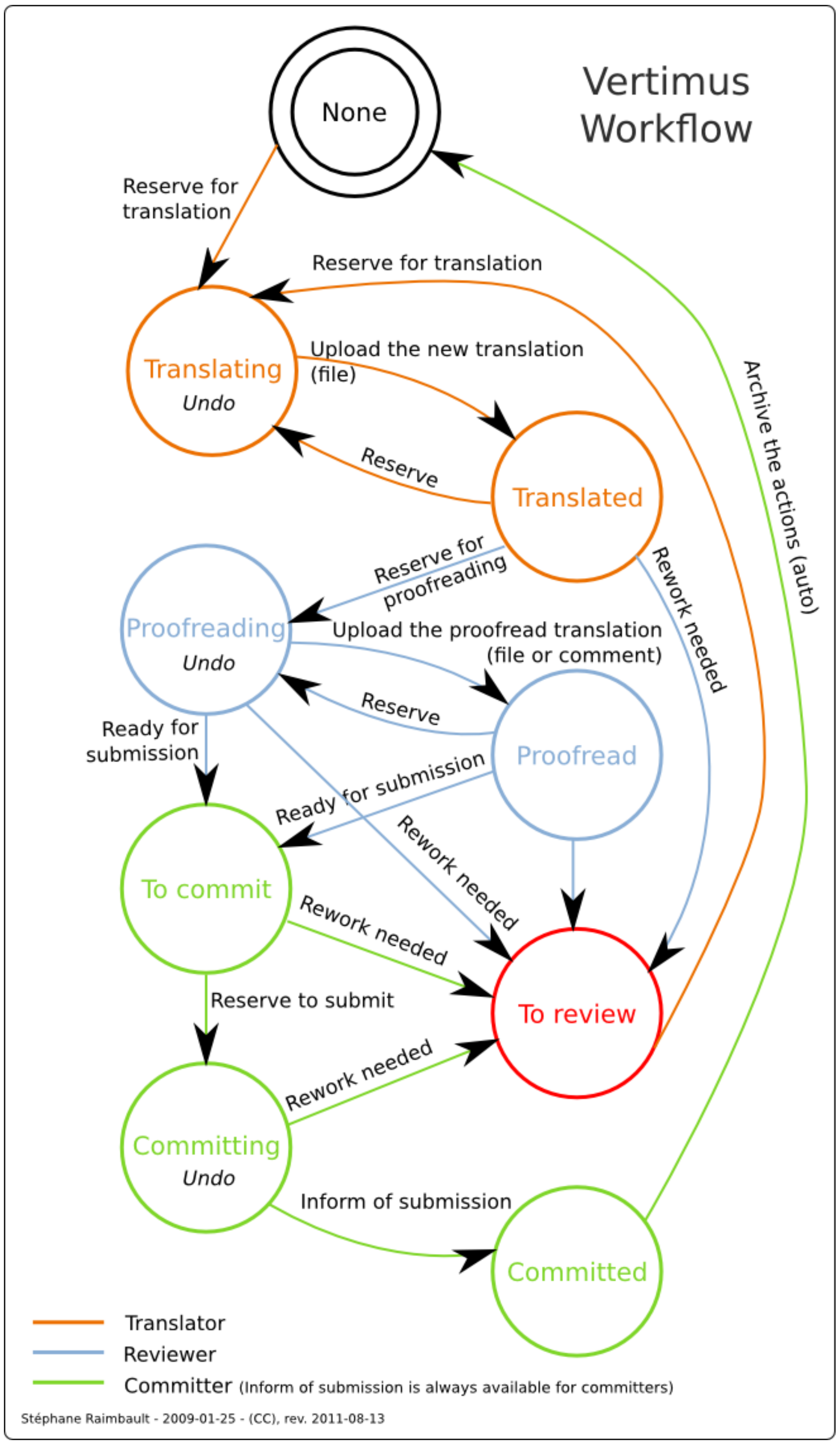
> *Reviewers are GNOME translators which were assigned by the team coordinator to review newly suggested translations (by translators, reviews or committers). They have access to all actions available to a translators with the addition of some reviewing task (ex reserve a translation file for proofreading, mark a translation as being ready to be included in GNOME).*
>
> *Committers are people with rights to make changes to the GNOME translations that will be release. Unless a translations is not committed by a committer, it will only remain visible in the web interface, as an attached PO file. Committers have access to all actions of a reviewer with the addition of marking a PO file as committed and archiving a discussion for new suggestions.*

The GNOME work-flow is characterized by everybody being able to suggest translations, by having a big body of people who can review those and by tightly controlling who can actually commit to the repositories. The possibility of reserving translations also minimize the chances of wasting time translating the same strings twice.

A very popular tool in the GNOME community of translators is the tool Poedit[40], though the work-flow does not encourage a specific tool for the translations themselves and GNOME translators do use several tools depending on their personal preferences.

This graph illustrates their work-flow:

[40] http://www.poedit.net/


Vertimus Workflow
None
Reserve for translation
Reserve for translation
Translating
Undo
Upload the new translation (file)
Reserve
Translated
Archive the actions (auto)
Reserve for proofreading
Rework needed
Proofreading
Undo
Upload the proofread translation (file or comment)
Reserve
Proofread
Ready for submission
Ready for submission
Rework needed
To commit
Rework needed
To review
Reserve to submit
Rework needed
Committing
Undo
Inform of submission
Committed
Translator
Reviewer
Committer (Inform of submission is always available for committers)
Stéphane Raimbault - 2009-01-25 - (CC), rev. 2011-08-13

505