Nautilus – Inside the Shell

Alexander Larsson - Red Hat, Inc

alexl@redhat.com

Table of Contents

1 Introduction

Nautilus is the official file manager and desktop shell for the GNOME desktop. This paper will give an overview of the design and implementation of Nautilus and some parts of the GNOME platform that it relies on. I will also give concrete pointers to interesting parts of the sources for developers who wish to hack on Nautilus, or just learn more about it.

This paper requires a basic understanding of how Nautilus works from the user side, and basic knowledge of GNOME programming.

2 The basic modules

Nautilus depends on most of the GNOME development platform, but there are some modules that are particularly important. This chapter will describe these modules and how they relate to Nautilus.

2.1 Gnome-vfs

Gnome-vfs is the basic I/O abstraction layer in GNOME. It implements a posix-like file API (open, read, write, close, readdir etc) that works on URIs instead of pathnames. The library is modular, so by writing modules that handle new types of URIs the number of protocols that gnome-vfs supports can be extended. Gnome-vfs ships with a few modules, including local files, ftp, http and ssh. Gnome-vfs-extras contains the popular smb method that allows you to read windows network shares.

The gnome-vfs file operations come in two flavours, synchronous and asynchronous. The synchronous versions work much like the normal posix calls, the call blocks while waiting for the result, which is then returned. The asynchronous versions are a bit different. Using these you register the operation you want to execute, and a function callback that you want called when the I/O has finished. Gnome-vfs then assigns the operations to a set of worker threads that execute the I/O, and wake up the main thread when done. In order to be able to respond to user input quickly and to never block the user interface Nautilus uses only asynchronous operations.

Gnome-vfs also includes the GNOME mimetype system. It contains code that can detect the mimetype of files, and a database that lets you map from mimetypes to the application that can handle it. It also allows for per-user overrides of this database so that the user can configure his own preferred application for different types of files.

2.2 Bonobo

Bonobo is the GNOME component model. Components in general are a way to declare public interfaces for well specified operations, so that objects implementing these can be found at runtime. This means you can update or even switch implementations of some services, without changing any code.

Bonobo in the GNOME 2 platform consists of three parts, bonobo-activation, libbonobo and libbonoboui. bonobo-activation is a database of installed components that allows you to write queries asking for components that satisfy some requirements. libbonobo is the user interface independent part of the component framework. libbonoboui contains user interface specific

parts of the framework, in particular the `BonoboControl` interface that allows you to embed a user interface object in an application. Controls use an xml-based description of the menus and toolbars of an application so that menu items and toolbar buttons from different components can be merged into the same set of menus.

Nautilus mainly uses Bonobo for the `BonoboControl` interface, in order to embed file and directory views into the Nautilus window. However, the metadata handling does use the user interface independent parts of bonobo.

2.3 EEL

EEL stands for “Eazel Extension Library”, and is really a part of Nautilus that has been split out as a separate module. It was originally part of Nautilus and called libnautilus-internal, but at some point in the development it was split out and renamed. This library contains utility functions and widgets that are useful to have when writing Nautilus, but that aren't really dependent on the environment of the Nautilus application.

This library is not in general meant to be used by other applications. It gives no backwards or forward compatibility guarantees and has no version management. It is meant to be upgraded in lockstep with Nautilus. However, sometimes code from EEL is moved to parts of the GNOME platform that other applications use. This happens when the code in question has shown itself stable and useful.

2.4 Nautilus

Nautilus itself is made up of several parts. Apart from the main executable there are executables and libraries for some of the components, and the libnautilus-private library that contains common functionality for the executables. There is also the public library libnautilus and its headers. It is used by third parties that want to implement nautilus extensions.

3 Source Tree Layout

3.1 EEL

All the code in EEL is in the `eel` subdirectory. Here is a description of some important files there:

`eel-accessibility.c:`
Utilities to help writing code for accessibility support.

`eel-art-extensions.c, eel-art-gtk-extensions.c, eel-glib-extensions.c, eel-gnome-extensions.c, eel-dateedit-extensions.c, eel-gconf-extensions.c, eel-gdk-extensions.c, eel-gdk-pixbuf-extensions.c, eel-gtk-extensions.c, eel-pango-extensions.c, eel-vfs-extensions.c, eel-xml-extensions.c:`
These contains helper functions that extend the API of some gnome platform libraries.

`eel-canvas.c, eel-canvas-rect-ellipse.c, eel-canvas-util.c:`
These are imported from the foocanvas module, which is a faster and simpler version of the Gnome canvas. It is used by the Nautilus icon view.

`eel-background-box.c, eel-background.c:`
Code to handle nice backgrounds with colors, images or gradients. This is used in places in Nautilus where the user can configure the background.

`eel-editable-label.c:`
A widget that allows editing of a multi-line single paragraph string. This is used when renaming files in the icon view.

`eel-ellipsizing-label.c:`
A version of label widget that ellipsizes the text. Ellipsizing means replacing part of the string with "..." in order to make it fit in the available area.

`eel-preferences.c, eel-preferences-glade.c, eel-enumeration.c:`
Functions for implementing preferences and preference dialogs.

`eel-i18n.c:`
Functions for string translation.

`eel-stock-dialogs.c:`
Functions to show some standard types of dialogs that Nautilus uses. Includes error and warning dialogs, timed dialogs (show up after a timeout and lets you cancel the operation) and helper functions to generate more general dialogs.

`eel-string.c:`
Some string helper functions

`eel-string-list.c`
A list of strings container, with lots of functionality.

3.2 Nautilus

The Nautilus code is split up into many directories. Here is an overview of what the different directories contain:

`libnautilus-private/:`
This is a library that is internal to Nautilus. It contains much of the core Nautilus functionality, such as metadata handling, volume management, trash handling, file operations, the icon container widget and drag and drop handling.

`libnautilus/:`
Contains the public library libnautilus, its headers and idl files. It is used by all Nautilus components, and it handles how views are connected to the Nautilus window and how components get integrated with the Nautilus menus and toolbars. The basic Nautilus component interfaces (`Nautilus::View` and `Nautilus::ViewFrame`) are described in the `nautilus-view-component.idl` file. The standard Nautilus menu and toolbar paths are listed in the `nautilus-bonobo-ui.h` file

`src/`
This is where the main application lives. Here is `nautilus-main.c` where execution starts, and here are important files like `nautilus-application.c` and `nautilus-window.c`. Much of the code here is about handling the various dialogs and windows that Nautilus show, and managing the menus and toolbars. Here we also have the code that selects and manages views when they are embedded in Nautilus windows (`nautilus-window-manage-views.c`).

`src/file-manager/`
Here lives the implementation of the default directory views, the icon view (`fm-icon-view.c`) and the list view (`fm-list-view.c`). These both inherit from a common class defined in `fm-directory-view.c`. The file `fm-properties-window.c` implements the file properties dialog.

`libnautilus-adapter/`
This library lets nautilus wrap ordinary bonobo controls as Nautilus views.

`components/`
This directory contains subdirectories for the components shipped with Nautilus. There are the components that implement sidebar tabs: `emblem`, `history`, `notes` and `tree`. `text` is the default view for text files. `throbber` contains the throbber used in the toolbar to show that Nautilus is working. `image_properties` contains a property page for image files. The code in `adapter` is used by libnautilus-adapter. `sample` contains a sample `NautilusView` that can be used as a starting point for new views. `loser` contains a test component that can be instructed to fail in various ways.

`cut-n-paste-code/libegg`
This directory contains code from the libegg library. The `update-from-egg.sh` script can be used to update the code to the latest version from libegg in cvs. Currently used code from libegg is: recent files, treeview multiple file dnd and per-screen help and exec functions (for multihead).

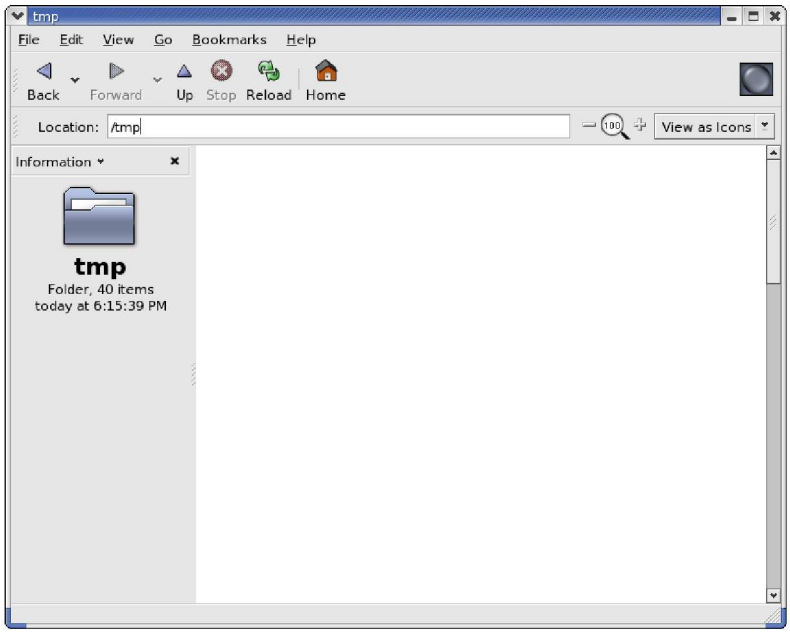
`cut-n-paste-code/widgets/gimphwrapbox`
The GtkWrapBox widget, copied from the Gimp.

`libbackground/`
This is really a separate cvs module, inherited into Nautilus using cvs magic. Its used to handle the desktop background settings.

4 The Nautilus Window exposed

4.1 NautilusWindow

The main Nautilus window class is `NautilusWindow`. It derives from `BonoboWindow` in order to allow it to use the Bonobo menu and toolbar merging code. A typical window looks like this:



A `NautilusWindow` is really just a shell. It has a user interface that consists of the standard toplevel menus and toolbars, including some operations like cut and paste, bookmarks, history and navigation. The user interface for this is specified in the `src/nautilus-shell-ui.xml` file.

The main purpose of the shell is to handle navigation between locations and to activate and manage the components used to show the current location. In order to actually show the current location the shell must locate and instantiate an appropriate component, and the component must communicate with the shell.

The interface used for this is defined in `libnautilus/nautilus-view-component.idl` and consists of two parts: `NautilusView`, and `NautilusViewFrame`. `NautilusView` is the interface the component implements. Its very simple, it just has a way to tell the component to start or stop loading a URI. `NautilusViewFrame` is the interface the shell implements. Every view must be placed in a view frame, and the component can use this interface to communicate with the shell.

In addition to this interface components use the libbonoboui interfaces to specify menus, toolbars and actions that are merged into the shell. This is how location specific operations are added to the shell, such as the ones related to file management.

In addition to the main view in the window there is a sidebar. The sidebar consists of a set of `NautilusViewFrame`s that load sidebar components. These components are activated when the window is created and the currently visible component can be selected with a menu at the top of the sidebar.

The desktop window is really just a normal window with everything but the main `NautilusView` hidden.

4.2 The NautilusView lifecycle

When a user tells the shell to load a new location, for example by entering something in the location bar, the shell must figure out how to display this location. Here are the steps that happen in this case: (Most of this happens in `nautilus-window.c`, `nautilus-window-manage-views.c` and `nautilus-applicable-views.c`)

1. The NautilusLocationBar fires the `location_changed` signal and the signal handler calls `nautilus_window_go_to`, passing it the URI to be opened.
2. The UI is set up for a load (stop enabled, throbber on) and the attributes for the target URI are requested using gnome-vfs.
3. The attributes of the file are used to figure out the default component to use for the URI. This is done by constructing a bonobo-activation query. Components are matched in the following order: component specified in the metadata for the location (set if the user visited the location with a non-default component before), default gnome-vfs component for the mimetype, gnome-vfs short list of components for the mimetype, components that lists the mimetype or the super-mimetype (e.g. image/*) in the `bonobo:supported_mime_types` attribute.
4. If the component selected is different than the currently loaded component the new component is loaded.
5. All views (main view and sidebar views) are told to load the location.
6. The view reports that loading is underway, and if we loaded a new main view component the old component is replaced by the new one.
7. The views periodically report their status, and then finally report that they are done.
8. The UI is set into a non-loading state. (Throbber disabled, stop disabled.)

When the view component is loading it will typically first merge in the specific menu and toolbar items that it supports, then it will schedule asynchronous gnome-vfs calls to load the file or directory, and then either wait until the file is loaded to display it, or incrementally display information while the data is being loaded.

4.3 The standard directory views

In order to act as a file manager the shell needs a view for directories. Nautilus ships with two views for directories, the icon view (`FMIconView`) and the list view (`FMListView`). The code for these views are in `src/file-manager`.

Both of these views inherit from `FMDirectoryView`, which has a lot of common code for handling directories. It merges in the `nautilus-directory-view-ui.xml` UI description, which contains all generic file operations like open, delete, rename, etc.

This directory also contains `FMPropertiesWindow`, which is the file properties window that both directory views can present, and `FMDesktopIconView` which is derived from the icon view adding support for desktop icons (trash, home, mounted volumes) and their

operations.

`FMIconView` merges some icon specific menu entries, and implements directory display as a set of icons with labels. The icon drawing is handled by the `FMIconContainer` class which is a subclass of `NautilusIconContainer` from libnautilus-private that just fills out the details related to file I/O and sorting.

`FMListView` implements a directory view as a list of icons, file names and file attributes. The widget itself is a `GtkTreeView` with a custom tree model for files (`FMListModel`).

5 The Nautilus I/O Model

5.1 NautilusFile

As explained above Nautilus uses asynchronous gnome-vfs operations in order to avoid blocking the main thread, causing an unresponsive user interface. It is not allowed to wait for a piece of information being read from a file without returning to the main loop to handle user input while waiting.

In order to make this easier the `NautilusFile` object has been introduced. Each `NautilusFile` object contains all the known information about a specific file, and when you want information about a file you just ask for the `NautilusFile` for it. Initially very little information is known about the file, just the name, and if you ask for some information about the object it will just return “I don't know”. This happens because `NautilusFile` isn't allowed to start a synchronous I/O request to read the information. Instead you have to request the information you need to be read.

There are two ways to request information about a file. If you just want to get the current information about a file, and don't care about future changes, you can call `nautilus_file_call_when_ready` with a `NautilusFileAttributes` bitmask stating the attributes you are interested in. The callback you supply will be called when the information you wanted is available in the `NautilusFile`. You just use the normal accessor functions to get the information. If the information is already known the callback will be called immediately, otherwise asynchronous I/O operations are scheduled to read it.

If you want to constantly show information about a file you instead use the `nautilus_file_monitor_add` call, stating the attributes you are interested in. You can then connect to the `changed` signal on the `NautilusFile`, which will be emitted every time some file information changes. Changes to files done from Nautilus will always be noticed, since all file operations in Nautilus are done through `NautilusFile`. If FAM is supported on the system modifications from the outside will also be noticed.

5.2 NautilusDirectory

Similarly to `NautilusFile` the `NautilusDirectory` object lets you handle directories of files. These work much like `NautilusFile`, you can add monitors to them, and the `files_changed` signal will be emitted whenever some file in the directory changes. You can also add a callback which will be called with a list of files when the directory has finished loading.

5.3 Caching data

The `NautilusFile` is the way Nautilus caches data about files. The information stored in the objects are kept until the last reference to the object is dropped. This means that the exact behaviour of Nautilus depends on whether someone keeps around a reference to the NautilusFile or not. Typically only the files in a directory visible in a Nautilus window are loaded in memory. If you really must have the latest information about a file you have to call `nautilus_file_invalidate_attributes` or `nautilus_file_invalidate_all_attributes` on the file to force Nautilus to forget the state of the file. If FAM is active this is generally not as big a problem, but remember that FAM doesn't work on all filesystems and different gnome-vfs methods.

6 Ways to plug in code to Nautilus

There are several ways to extend the capabilities of Nautilus without directly modifying the code. This way you can implement new functionality or integrate other applications with Nautilus.

6.1 VFS module

The most low-level way to extend Nautilus is by writing a new gnome-vfs module. This extends the set of protocols and filesystems that Nautilus and other GNOME applications can read files from. When doing this you can't really implement specific user interface for your extension, because gnome-vfs is just a filesystem API. However, all GNOME applications can immediately use your module to load and save files.

6.2 NautilusView

The most generic way to extend Nautilus is to write a new `NautilusView`. The view can specify what mimetypes it supports, and what URI schemes it supports. This way you can write both viewers for particular types of files and for directories. The directory viewers can even be set up to be used only when there are files of a particular type in the directory.

To implement a new `NautilusView` is quite a lot of work, since the view has to do everything related to I/O and widget rendering. Views distributed outside Nautilus cannot use libnautilus-private, and therefore not e.g. `NautilusFile`, since this is private unstable API. However, the complexity of writing a `NautilusView` can pay of since you can present the information you want in exactly the way you want.

6.3 Context menu item

If you want to integrate an application with the file manager you can install a context menu component for a particular file type. This will add a menu item in the context menu for files in the icon and list views that lets you launch your application on the file(s). The name of the menu item can be translated.

This works by installing a component in bonobo-activation that has a boolean attribute called `nautilus:context_menu_handler` set, and optionally a `nautilus:can_handle_multiple_files` attribute. When the other normal attributes of the component (`bonobo:supported_mime_types`,

`bonobo:supported_uri_schemes`, etc) matches the files selected when the context menu is showed, all the attributes starting with `nautilusverb:` are read and put into the context menu.

The attributes can look like this: (the attributes with underscores will be translated by intltool): `<oaf_attribute name="nautilusverb:DoExtract" type="string" _value="Extract To..."/>`

When the context menu item is selected the component will be activated, and the `Bonobo::Listener::event()` method will be called with a sequence of the selected URIs as arguments.

Examples of how to implement context menu plugins are available in file-roller, nautilus-cd-burner and fontilus.

6.4 Property page

Applications can install custom notebook pages for the file properties dialog. These can show and/or edit properties of files in a way that are more appropriate for some specific type. This is done by installing a component that implements `BonoboControl` with the string attribute `nautilus:property_page_name` specified. The string is used as the name of the tab, and can be translated. The normal Bonobo attributes like `bonobo:supported_mime_types` are used to specify what files the page should be used for.

The Nautilus source contains the image properties component which is a good example of how to write property pages. It is located in `components/image_properties`.

6.5 Side bar view

Just like you can implement main `NautilusViews` you can also implement custom views for the sidebar. These are normal `NautilusViews` that has the `nautilus:sidebar_panel_name` property set. The value of this property is the name used in the sidebar menu.

In addition to the normal `NautilusView` interface the component can also set the `tab_image` property on the `BonoboControl` property bag in order to show an icon in the sidebar. See the notes sidebar in the Nautilus code for an example of this.

6.6 Mime handler

The easiest way to integrate with Nautilus is to just install gnome-vfs mime data files so that the files of the type you handle are recognized by GNOME as handled by your application.

7 Various Objects/subsystems

This section will describe various important subsystems in Nautilus that are used internally. They are all in libnautilus-private, so only Nautilus code can use them, and the APIs are subject to regular changes.

7.1 NautilusIconFactory

The icon factory is the icon lookup and cache system in Nautilus. It works in two steps, first it maps from a `NautilusFile` to an icon name, then it uses the current icon theme to look up the image file.

The mapping is done using `nautilus_icon_factory_get_icon_for_file()` which looks at the type and state of the file to generate the icon. If there is a thumbnail stored for the file or a specific icon stored in the file metadata the absolute filename to this file is returned instead of the icon name.

The second step is to get a pixbuf representing the icon. This is accomplished by calling `nautilus_icon_factory_get_pixbuf_for_icon()` which returns a reference to a pixbuf which is also stored in the icon cache. It also returns various information about the icon, like where to embed text in it and where to position emblems.

There are also convenience functions to do both steps in one go, and some functions to support emblem icons.

Under the covers all the mapping, thumbnail handling and icon theme support is done by calls to `libgnomeui`, so other applications can use these to get the same icons for files that Nautilus does. The icon factory is just a convenient wrapper and cache for these that fit well in the `NautilusFile` system.

7.2 NautilusIconContainer

`NautilusIconContainer` and `NautilusIconCanvasItem` make up the widget that the icon view uses to render the view. It is derived from `EelCanvas` which is an imported copy of the `FooCanvas` module. This is a simplified, faster version of the gnome canvas.

The container is abstracted out from the I/O model, and you need to derive from it to add the functionality needed to show file data. This is done by the `FMIconContainer` class in Nautilus.

The container implements all the functionality needed by the icon view, such as icon scaling, renaming, text embedding in icons, different forms of icon layout, drag and drop, and zooming.

7.3 NautilusVolumeManager

The `NautilusVolumeManager` object (there is only one) is what keeps track of the mounted filesystems in the system, and the available removable media volumes. It periodically looks for changes in the system mtab and updates its internal tables. If you are interested in changes to the mounts you can connect to the `volume_mounted` and `volume_unmounted` signals.

Mountpoints in the system are exposed using the `NautilusVolume` type, which allow you to quickly look at the type of the mount, mount path, device path and other information. You can also get a list of all the removable media volumes in the system, enumerate all the currently mounted volumes and mount/unmount/eject removable volumes.

One issue to be wary of in the volume monitor is that the `NautilusVolume` objects are not persistent between reads of the mtab file, so you can't keep around references to them.

7.4 Thumbnailing

The Nautilus thumbnail system consists of a priority queue where `NautilusFile` objects that should be thumbnailed are stored and a separate thread that keeps generating thumbnails for the first file in the queue. When the thumbnail is finished the `changed` signal on the `NautilusFile` is emitted and the views note that the icon must be redrawn. When the icon for the file is looked up again, the icon factory will notice the newly generated thumbnail and use that.

There are functions to move thumbnails to the top of the priority queue and to remove them from the queue. The views uses these in order to prioritize the icons currently visible on the screen, and to avoid thumbnailing files no longer in an active window.

7.5 Metadata

Nautilus can store metadata information about files. This information is stored as an xml file per directory in `$HOME/.nautilus/metafiles`. At some point we hope to migrate the metadata system to gnome-vfs, but currently it can only be used by Nautilus.

The metadata system is implemented as a Bonobo server that handles all accesses to the metadata files. This server lives inside the first Nautilus process that is started by the user (unless it exits with no other users), and all other Nautilus processes talks to it. This means metadata is always synchronized between different instances of Nautilus and out of process components. Changes to metadata are sent to all objects that monitor the metadata for the directory.

When using metadata inside Nautilus all you have to do is use the `NautilusFile` API related to metadata. It allows you to get and set metadata of various types, and if you monitor the file, changes to metadata will cause the `changed` signal to be emitted on the file, and the `files_changed` signal on the directory.

7.6 Trash

Nautilus uses the gnome-vfs trash system. It works by creating a trash directory for each mount point, when it can. The directory is called `.Trash-username` and stored in the top directory of the mount. For files on the same device as the user homedir `$HOME/.Trash` is used instead.

In order to make all these directories understandable for the user Nautilus introduces the fake URI `trash:`, which contains the merged information from all the trash directories. The trash directories are managed by the `NautilusTrashMonitor` object. When new trash directories are discovered it emits the `check_trash_directory_added` signal. These signals are caught by a special type of `NautilusDirectory` called `NautilusTrashDirectory`. This object handles the merging of the real `NautilusDirectory` objects for the trash directories, and corresponds to the `trash:` location.

The trash monitor also emits the `trash_state_changed` signal whenever the trash state

changes from empty to full or back. This is used to pick the right icon for the trash desktop icon.

7.7 Preferences

Nautilus uses GConf to store its preferences, but in order to make handling preferences easier there are some utilities in EEL to define and use application preferences. There are functions to make it easier to write preferences dialogs, and to connect preferences to glade files.

The file `libnautilus-private/nautilus-global-preferences.h` lists all the application preferences in Nautilus, and to read them you just have to call `eel_preferences_get()` or a similar function.

There are also support for automatic preferences. You just give the name of the preference and a location to store it in, typically a static global variable. EEL will then update the variable whenever the preference value changes, so you can just access the global variable in the code.