Lübeck University
Institute for Theoretical Computer Science

Studienarbeit

Migrating the Thunar File Manager to the Extensible Asynchronous Virtual File System Layer GIO

Jannis Pohlmann

October 1, 2009

Supervised by
Prof. Dr. Till Tantau

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Lübeck, October 1, 2009

Abstract

Contents

Listings ix

1 Introduction 1
1.1 Thunar and GIO in a Nutshell . . . 2
1.2 A Brief Comparison of ThunarVFS and GIO . . . 4
1.3 Overview of the Migration to GIO . . . 4

2 Migration Process and Selected Topics 7
2.1 Planning Phase and Migration Strategy . . . 7
2.2 A Detailed Comparison of ThunarVFS and GIO . . . 8
2.3 Resolving and Loading Files Asynchronously . . . 14
2.4 Framework for Time-Consuming Operations . . . 18
2.5 Thumbnail Generation Using the Tumbler D-Bus Service . . . 20

3 Summary and Evaluation 29
3.1 Summary of the Changes . . . 29
3.2 New Features . . . 30
3.3 Known Major Regressions . . . 33
3.4 Consequences for Other Applications . . . 34

4 Outlook 35

Literature 37

Listings

2.1 The ThunarBrowser methods for resolving files and volumes. 16
2.2 Simple use case of the ThunarBrowser methods. 17
2.3 Declarations of ThunarVfsJob and ThunarVfsSimpleJob in simplified Vala syntax. 19
2.4 API of ThunarVfsThumbFactory in simplified Vala syntax. 21
2.5 Example of a custom ThunarVFS thumbnailer definition. 21
2.6 D-Bus interface definition for the thumbnailer service. 23
2.8 Output of the example thumbnailer client. 25
2.7 Example thumbnailer client written in Vala. 26

1 Introduction

Virtual file systems allow applications to access different types of file systems in a uniform way. They are responsible for providing the following features to other libraries and applications:

- a uniform way to access different (virtual) file systems;
- detailed information about files and directories;
- high-level APIs for operations like copy, move, rename, trash or delete;
- ways to monitor files and directories for changes;
- an association of files with applications installed on the system; and
- storage device handling, also referred to as volume management.

Among the variety of applications mostly using virtual file systems in order to support as many different locations as possible for writing and reading, file managers are probably the most extensive and complex example for their use.

This thesis describes the migration of the open source file manager Thunar from its original virtual file system (VFS) implementation ThunarVFS to an asynchronous and extensible implementation called GIO. Various migration and implementation problems are covered in more detail as they might be of wider interest when writing file managers or desktop applications or when dealing with asynchronous application programming interfaces (APIs) in general.

The term virtual file system is often used ambiguously and this thesis is no exception. It is applied to abstraction layers in the same manner as it is used for describing concrete implementations of file system backends. The most prominent example used in the following chapters involves GIO (the GLib Input, Output and Streaming Library) and GVfs, a GIO extension and collection of several backends for different virtual and remote file systems.

This chapter introduces Thunar and GIO by describing their origin and context as well as the design of the file manager in section 1.1. For a better understanding of the general purpose and architecture of ThunarVFS and GIO, a basic comparison is given in section 1.2. An in-depth comparison with special focus on technical aspects can be

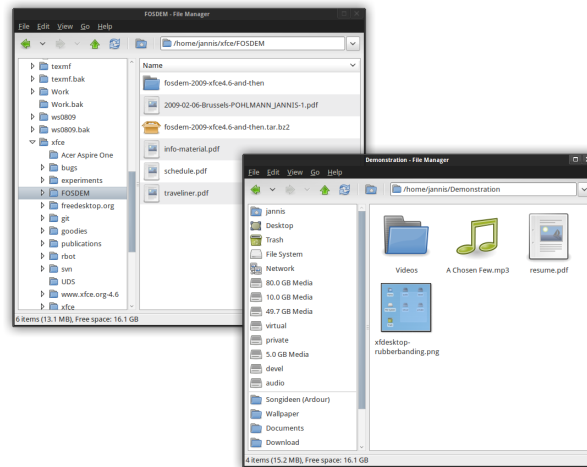

Figure 1.1. Thunar using different views and toolbars

found in chapter 2. The chapter closes with section 1.3 by providing a quick overview of the migration and the topics selected for this thesis.

1.1 Thunar and GIO in a Nutshell

Thunar is a file manager written and maintained as part of the open source desktop environment Xfce. Originally started by Benedikt Meurer in 2005, it has been around since the 4.4 release of Xfce in late 2007. Thunar was designed from the ground up to be fast and intuitive. It can be used with assistive technologies and implements the most relevant freedesktop.org specifications to ensure compatibility with other desktops and applications. As a file manager its main task is to enable users to browse and manage files and folders on their computers. This includes everything from listing folder contents, launching copy, move and delete operations and opening files with applications installed on the system up to generating preview images and handling storage devices, often referred to as volume management. In a modern desktop scenario, the file manager is usually also responsible for supporting other applications in accessing the file system by providing commonly used services.

Like in all other components of Xfce, the GTK+ toolkit is used to build the graphical user interface of Thunar. Even though being written in C, Thunar follows the object-oriented programming paradigm by making heavy use of GObject and the GType type system, both of which are part of GLib, a utility library that is also used by GTK+ and the rest of Xfce. Thunar is split up into three components, as can be seen in figure 1.2. As mentioned briefly before, ThunarVFS is a file system abstraction layer which

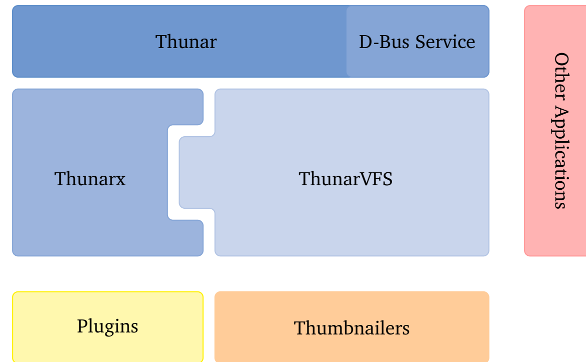

Figure 1.2. Basic architecture of Thunar

was developed together with Thunar. Being the library to be migrated away from, it plays a major role in this thesis. ThunarVFS is described in more detail in sections 1.2 and 2.2. Thunarx is an extension framework which allows plugins to be written for the file manager. Finally, Thunar itself describes the application written on top of these libraries, which aside from the graphical user interface also provides a D-Bus (a widely used inter-process communication system) service for other desktop applications.

GIO was, along with GVfs, developed by Alexander Larsson with the goal to eventually replace GnomeVFS, a VFS layer previously used inside the GNOME desktop environment. GnomeVFS had numerous shortcomings which were discussed broadly [5]. GIO avoids these limitations by providing a high-level asynchronous API that is heavily extensible. It was merged into GLib in late 2007 and was first released as part of GLib 2.16.0 in March 2008.

The reason for moving the virtual file system layer to the lower bottom of the GTK+ software stack was to enable GTK+ and all applications built on top of it to take advantage of it. Designed for extensibility, it mainly serves as a uniform wrapper for various VFS backends. These can be loaded on demand, typically run in separate processes and communicate with GIO over D-Bus. They can thus be easily shared among all applications in the current desktop session—something from which the user experience greatly benefits. Several such implementations are available as part of GVfs, the most accepted set of GIO extensions to date. In order to be useful on its own, a virtual file system extension for accessing local files is already shipped with GIO itself.

1.2 A Brief Comparison of ThunarVFS and GIO

Both, ThunarVFS and GIO are virtual file system abstraction layers. As such they provide roughly the same set of features that was already mentioned earlier in the introduction. In file managers virtual file systems like ThunarVFS and GIO play a key role as every piece of information is retrieved and all operations on the real file system are performed through them.

The main differences between ThunarVFS and GIO are the extensibility and asynchronous API design of GIO. Unlike ThunarVFS, which was written especially for being used in Thunar, GIO takes a much more generic approach and leaves a lot of implementation details open to extensions. Extension points are available for virtual file systems such as the SSH File Transfer Protocol or Windows network shares, volume management, file monitoring and the association of file types with applications. This makes it much more flexible and powerful than ThunarVFS.

GIO provides asynchronous APIs for all operations that may run for a long time, and, if performed synchronously, may freeze the user interface of an application. This includes everything from querying file information up to mounting volumes. Asynchronous functions are very important when dealing with slow drives and remote file systems. Since ThunarVFS only supports local files, it is only logical that most of its APIs (except for file operations) are synchronous.

1.3 Overview of the Migration to GIO

Abandoning a virtual file system layer in favor of another in a file manager and an entire desktop environment is nothing short of a complex undertaking. As can be deduced from the comparison in section 1.2, everything a file manager does is based on the information it requests from and the operations it performs on the virtual file system. Other desktop applications depend on the file manager as a service for common routines and graphical user interface elements like file information dialogs. Changes made to the foundations of a file manager not necessarily but often affect its behaviour and thus, have implications on other applications as well. This especially holds when modifications introduce gaps in functionality. Finally, abandoning a virtual file system layer that is directly used in other applications also involves migrating these applications.

The migration of Thunar to GIO is motivated in various ways. First of all, Xfce is a very small open source project with only about half a dozen core developers. Every new component introduces a new maintenance burden. Like many companies that are trying to outsource development and maintenance of open source software pieces into different

communities (Nokia does this with parts of their Maemo platform for instance), one of the goals of Xfce is to reduce the overall maintenance overhead in order to concentrate on the core of the desktop by to reusing components developed and maintained by others—especially when they are already part of the software stack being used, as is the case with GIO. Being part of GLib and thereby widely used and tested in GTK+ and GNOME, GIO is very likely to be maintained for a long time. Thus, it is reasonable to adopt it and only maintain the pieces missing due to gaps in functionality compared to ThunarVFS.

If maintenance by itself already is a high burden, as is the case with ThunarVFS, this holds even more for the development of new features. ThunarVFS, except for thumbnailers, is not extensible at all. As a consequence its range of features is limited by the amount of direct contributors to ThunarVFS. GIO on the other hand provides many extension points which makes it possible to incorporate features written by other parties without having to maintain them. The most important extension point is the one for `GFileInfo`, allowing virtual file system backends for arbitrary URI schemes such as `sftp://` or `computer://`. This is one of features most desired and requested by users. With the availability of extension bundles like GVfs, GIO introduces a number of new features, such as a better integration into Windows networks, eventually leading to a better user experience.

The migration is also personally motivated. Performing such a complex migration—the Thunar application (figure 1.2) alone is worth about 58,000 physical lines of code—is a real challenge. Keeping the entire application in a working state throughout the entire step-by-step process introduces additional difficulties. Last but not least, basing one's work on an existing and complex code base trains software development skills and helps in learning more about software design and refactoring.

The following chapters describe the migration process as well as its results and consequences. The most interesting and universal topics of the migration are covered in more detail. The results are briefly evaluated in chapter 3 by summing up what new features and regressions were introduced and how other applications are affected by the changes made. Finally, an outlook is given in chapter 4 with a special focus on how the situation after the migration can be further improved.

2 Migration Process and Selected Topics

2.1 Planning Phase and Migration Strategy

A migration as complex as this has to be planned thoroughly. In this particular case, the first thing to investigate was which components were directly involved in or affected by the migration of Thunar to GIO.

Thunar The file manager itself. Uses ThunarVFS for almost everything. Needs a few fundamental changes to work with GIO.

Thunarx The Thunar extension library. Overlaps with ThunarVFS only in a single function. Easy to migrate.

ThunarVFS The current VFS layer used by Thunar. Ideally moved into its own package for other applications to depend on it until they have been migrated.

Thunar Volume Monitor An application to perform actions when devices are plugged or unplugged. Integral part of the user experience concept of Thunar. Uses ThunarVFS only for monitoring its configuration file.

Thunar Extensions There are six extension packages for Thunar, plus a few plugins shipped with Thunar itself. Most of them use ThunarVFS for querying file information and will have to be migrated.

Thunar Thumbnailers A set of scripts called by ThunarVFS to generate preview images for files. As GIO does not implement thumbnailing itself, it would be nice to provide a solution that works without ThunarVFS but still supports these scripts.

Xfce Desktop Manager An Xfce core component responsible for managing the desktop, including file and device icons. Relies on Thunar for file operations and ThunarVFS for thumbnailing and file information. Has to be migrated to GIO. Ideally, the D-Bus service API provided by Thunar would be extended to re-use Thunar widgets like the file properties dialog.

GIO The future replacement for ThunarVFS. Its differences compared to ThunarVFS, including advantages and shortcomings, are discussed broadly in sections 1.2 and 2.2.

GVfs Besides shipping VFS backends for various URI schemes, also provides a volume management implementation which is not part of GIO itself. Unfortunately depends on several GNOME modules. To maintain feature parity before and after the migration without drastically increased dependencies, a separate volume management implementation would have to be written.

Some of these components, including several of the extensions and the Xfce desktop manager, are out of the scope of this thesis, as they are maintained by other people. The focus is thus set on replacing all occurrences of ThunarVFS inside the Thunar and Thunarx source code with (hopefully) equivalent code based on GIO. As a step-by-step documentation of this process would make for a very long and tedious piece of paper, section 2.2 instead concentrates on explaining the implementation and API differences of ThunarVFS and GIO. Descriptions of the resulting search-and-replace work and minor refectorings are left out for obvious reasons.

As we will see, some of the differences of ThunarVFS and GIO raise more problems than others. Selected differences that might have a wider impact on the desktop as a whole are highlighted in sections 2.4, 2.5 and 2.3.

2.2 A Detailed Comparison of ThunarVFS and GIO

In the following, ThunarVFS and GIO are compared by describing how they implement the VFS-typical features listed in section 1.2, and also how they differ from one another, both from a technical and a user point of view. The implications on the migration—or to be more specific, the modifications required to port the majority of the Thunar source code to GIO—are not explained in detail. As we will see, most of the differences presented here are of minor scale, and thus, almost trivially resolvable. The ones which have a major impact are described later in this chapter.

Accessing Different File Systems

Files on supported file systems are addressed either by local paths or URIs. ThunarVFS only implements the `file://` and `trash://` URI schemes. GIO by default only supports `file://` but can be extended to support arbitrary schemes. Through GVfs, `trash://`, `sftp://`, `ftp://`, `network://` and several others are available.

Since path or URI strings are rather inconvenient to work with, both, ThunarVFS and GIO provide the `ThunarVfsPath` and `GFile` wrapper classes for defining and working with URIs. They provide about the same functionality and thus, one can easily replace the other. Their most important features include parsing local paths, URIs and command

line arguments and computing how two files are related to each other in the file system hierarchy (e.g. via an ancestor–descendant relationship).

Querying File Information

For loading information about files and directories, again, ThunarVFS and GIO have two very similar classes called `ThunarVfsInfo` and `GFileInfo` of which the latter is actually an interface that can be implemented by extensions in order to add support for other virtual file systems. They are created from a `ThunarVfsPath` and `GFile` with the main difference being that `GFileInfo` allows to select which bits of the overall file information are loaded. Other than that they are more or less equivalent. In ThunarVFS, the size of a file can be queried with the following lines.

```
ThunarVfsPath *path = thunar_vfs_path_new (uri);
ThunarVfsInfo *info = thunar_vfs_info_new_for_path (path, &error);
ThunarVfsFileSize size = info->size;
```

The same code based on GIO looks slightly more complicated but is also more powerful. The following example also demonstrates how only certain attribute names like `standard::size` or attribute namespaces like `standard` can be selected.

```
GFile *file = g_file_new_for_uri (uri);
GFileInfo *info = g_file_query_info (file, "standard::*", cancellable, &error);
guint64 size = g_file_info_get_attribute_uint64 (info, "standard::size");
```

Filtering attributes can have positive impacts on the performance when operating on slow drives or remote file systems. However, due to the internal design of Thunar, information about a file is usually only queried once. Thus, being able to filter means to make a decision on what to filter. This decision would be easy if it only affected the file manager itself. Unfortunately, plugins written based on Thunarx by default only have access to the same information Thunar is loading. They can query their own file attributes but the the information which attributes and namespaces are initially available to plugins has to be defined somewhere in the Thunarx API. At the time of writing, the namespaces `access`, `id`, `mountable`, `preview`, `standard`, `time`, `thumbnail`, `unix` and `filesystem` are being used.

File Operations

File operations like copy, move, rename, trash or delete are handled differently in ThunarVFS and GIO. GIO provides one synchronous and asynchronous method for each

operation to which a finish and sometimes a progress callback can be passed. In addition, these methods optionally take a `GCancellable` parameter in case the operation can be cancelled by the user from the graphical user interface. The following example illustrates how a file copy operation can be triggered with GIO.

```
GFile *source = g_file_new_for_path ("/tmp/source-file");
GFile *destination = g_file_new_for_path ("/tmp/target-file");
g_file_copy_async (source, destination, G_FILE_COPY_NONE, 0, cancellable,
                   progress_callback, user_data, finish_callback, user_data);
```

The operations supported by GIO only work on individual files and are non-recursive. For this purpose the API is decent. However, it is insufficient for complex operations like copying or moving directories recursively.

ThunarVFS on the other hand provides a generic and much more versatile framework for asynchronous and possibly interactive operations, which is implemented via several GObject classes called `ThunarVfsJob`, `ThunarVfsSimpleJob` and `ThunarVfsTransferJob`. Instead of passing callbacks to a function, one can create a `ThunarVfsJob` object and connect to signals with different meaning. This is a concept that is used everywhere in Thunar, GTK+, and basically everything based on GObject, and thus fits well into the design of many applications.

ThunarVFS also offers very powerful functions for performing recursive and non-recursive file operations such as directory listings, copying, moving, deleting files and directories and others like changing permissions recursively, all based on `ThunarVfsJob`. An example usage of this is demonstrated here, again initiating a file copy operation.

```
ThunarVfsPath *source = thunar_vfs_path_new ("/tmp/source-file", NULL);
ThunarVfsPath *destination = thunar_vfs_path_new ("/tmp/target-file", NULL);
ThunarVfsjob *job = thunar_vfs_copy_file (source, destination, &error);
g_signal_connect (job, "percent", G_CALLBACK (progress_callback), user_data);
g_signal_connect (job, "finished", G_CALLBACK (finish_callback), user_data);
g_signal_connect (job, "ask-replace", G_CALLBACK (ask_callback), user_data);
```

While this looks more complicated and verbose than the GIO snippet on the first sight, it is much more flexible and in line with the coding style being used in the rest of Thunar. Due to its important role inside Thunar and also in the migration, the job framework is covered in more detail in section 2.4.

File Monitoring

ThunarVFS and GIO allow individual files and directories to be monitored for changes using dedicated classes. In ThunarVFS there is a singleton class called `ThunarVfsMonitor`

with `thunar_vfs_monitor_add_file` and `thunar_vfs_monitor_add_directory` functions to set up monitoring. These functions take a `ThunarVfsPath` and a monitor callback and return a unique identifier with which the monitoring can be cancelled again later.

GIO has an interface similar to this called `GFileMonitor`. Its implementation may vary depending on the platform but unlike `ThunarVfsMonitor` it is not necessarily implemented as a singleton. File monitoring can be set up with the `g_file_monitor` function which takes a `GFile` pointing to a regular file or a directory, and returns an instance of `GFileMonitor`. One can then connect to a change signal of this instance to be notified of changes made to the file. Once monitoring is set up the way described above, the workflow resulting from the two APIs is again very much the same.

Association of Files With Applications

On the Linux desktop files are, by tradition, associated with applications via their MIME type, a two-part identifier for file formats originating from Multipurpose Internet Mail Extensions standard for non-ASCII character sets in emails, non-text attachments and multi-part message bodies [3].

The problems how to determine the MIME type of a file and how to register an application as a possible MIME type handler are addressed by two freedesktop.org specifications [6] [1]. In addition to storing MIME types supported by an application in its desktop entry, the file `$XDG_DATA_DIRS/applications/mimeinfo.cache` can be used to define additional MIME types for an application. The association of MIME types with default applications aare then defined using the files `$XDG_DATA_DIRS/applications/defaults.list` (for global defaults) and `$XDG_DATA_DIRS/applications/mimeapps.list` (individual users). It is noteworthy that `mimeinfo.cache` and `defaults.list` are not part of any freedesktop.org specifications and, for instance, not used by the KDE desktop environment. Based on this information, ThunarVFS and GIO implement various classes and functions to determine the MIME type of a file, query applications capable of opening a file, get and set the default application for a type and launch files with applications. The end result can be seen in figure 2.1.

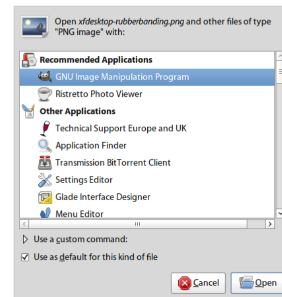

Figure 2.1. Associating files with applications in Thunar.

In ThunarVFS, each file is associated with a `ThunarVfsMimeInfo`, which is a simple wrapper around its MIME type string. Using this information, applications able to handle the file can be queried from a singleton `ThunarVfsMimeDatabase`. Apart from the class

ThunarVfsMimeApplication which represents applications, ThunarVFS also supports a deprecated but very useful part of the desktop entry specification [1] called desktop actions. They allow desktop entries to include a number of actions in addition to the information about the application itself. These actions, 'Burn image' for instance, too can be connected to MIME types and then be used in context menus. In ThunarVFS, they are implemented via the ThunarVfsMimeAction class. To unify the workflow with these two classes, there is an interface called ThunarVfsMimeHandler which both classes implement. The following example demonstrates how these APIs can be used to determine the default application for a file.

```
ThunarVfsInfo *info = thunar_vfs_info_new_for_path (path);
ThunarVfsMimeDatabase *db = thunar_vfs_mime_database_get_default ();
ThunarVfsMimeApplication *app =
  thunar_vfs_mime_database_get_default_application (db, info->mime_info);
```

GIO drastically simplifies this. There is no wrapper for MIME types. Instead, each GFileInfo has a standard::content-type string attribute. There is a number of utility functions to work with these strings, like g_content_type_equals. To represent applications in the code an interface called GAppInfo is used. It is not limited to the aforementioned specifications but has an implementation based on desktop entries built in. GAppInfo can be used in a similar fashion as the APIs from ThunarVFS, as is shown in the following code snippet.

```
GFileInfo *info = g_file_query_info (file, ...);
const gchar *type;
type = g_file_info_get_attribute_string (info, "standard::content-type");
GAppInfo *app = g_app_info_get_default_for_type (content_type, FALSE);
```

Volume Management

Storage device handling, also known as volume management or volume monitoring, is implemented very differently in ThunarVFS and GIO. Traditionally, volume management implementations on Linux are written on top of a hardware abstraction layer creatively called HAL. Since 2008 the trend is to abandon HAL in favor of DeviceKit, a modular system service with D-Bus interfaces for storage devices, power management and more. DeviceKit-disks provides the D-Bus interface for volume management. Like HAL, DeviceKit was first written for Linux and is now slowly being ported to other Unixes.

ThunarVfsVolumeFreeBSD and ThunarVfsVolumeHal. With HAL undergoing deprecation, this would soon require a rewrite based on DeviceKit-disks. Overall, the API

presented to users of the library is very simple. There are only two classes, of which the first one, `ThunarVfsVolumeManager` provides methods to query available volumes. The returned volumes are instances of `ThunarVfsVolume` and can be used to get information about a CD-ROM drive, floppy disk, USB stick, memory card or partition. `ThunarVfsVolume` also offers methods to eject, mount and unmount a volume and signals for when a volume is mounted or unmounted. All of these methods are synchronous and very straightforward.

ThunarVFS has two different volume management implementations built in:
The equivalent GIO API is once again a collection of interfaces which can easily be implemented in the form of GIO extensions. Implementations based on HAL and gnome-disk-utility, which is an abstraction layer on top of DeviceKit-disks, are available as part of GVfs. GIO exposes much more of the underlying hardware. In addition to the `GVolume` interface which represents user-visible partitions, there are separate interfaces for physical drives and user-visible mounts called `GDrive` and `GMount`. `GDrive` and `GMount` are only really needed for checking whether a partition is already mounted or not. `GVolume` is the main interface used to query information about partitions and to mount and unmount them. A list of all available volumes can be retrieved through `GVolumeMonitor` which is equivalent to `ThunarVfsVolumeManager` in that regard.

Another difference between ThunarVFS and GIO here is that, once again, all possibly long-running or interactive operations like mounting or ejecting are asynchronous and care has to be taken of side effects like race conditions when using them.

Other Notable Differences

File managers, desktop managers, image viewers and a lot of other applications often require preview images of files, also referred to as thumbnails, to make them more easily distinguishable or to add a visual aspect to the usually mostly textual file information. Files for which thumbnails are often required include image files, videos, PDF or word documents and fonts. What thumbnail formats are generated and how and where they are stored is defined by the thumbnail managing standard [2], another freedesktop.org specification.

GIO only has rudimentary support for thumbnailing. When a `GFileInfo` is queried with the `thumbnail::path` attribute enabled it checks whether a valid thumbnail file exists and if so, sets the attribute value to the path of the thumbnail. There is no API to create or manage thumbnails.

ThunarVFS on the other hand not only provides methods to query, generate and store thumbnails via the `ThunarVfsThumbFactory` class. For once, it also offers an ex-

tension point for custom thumbnailer scripts. Thumbnailers are defined using desktop entries [1] and can be used to generate thumbnails for literally any MIME type. ThunarVFS itself ships thumbnailers for JPEG, image formats supported by GdkPixbuf, an imaging library used by GTK+, as well as fonts. Thumbnailers for PDF, videos and more are provided by a separate component called thunar-thumbnailers.

The lack of thumbnailing support in GIO raises the problem how to deal with the thumbnailing code from ThunarVFS with the ultimate goal to maintain the same feature in Thunar even after the migration. Section 2.5 discusses possible solutions and the final implementation in detail.

2.3 Resolving and Loading Files Asynchronously

The asynchronous GIO APIs for loading files, mounting volumes and the support for remote file systems have implications on how an application has to load files and directories. Before accessing files on a volume, the volume first needs to be mounted. Some files are only shortcuts to other files and have to be resolved before opening them. There are also files that can be mounted themselves, similar to volumes. The application preferably has to handle all this in a unified way to avoid redundant source code. In this section, these special cases are covered in more detail, followed by a presentation of a unified solution implemented for Thunar.

Mounting Volumes

File information queries on files on volumes that are not yet mounted into the system raises errors. Thus, before accessing such files, their enclosing volume have to be mounted first. Whether or not the enclosing volume of a file is mounted can by checked with a call to `g_file_query_info`. If the returned error code is `G_IO_ERROR_NOT_MOUNTED`, the volume needs to be mounted before any file information can be loaded. This can only be done asynchronously by calling `g_file_mount_enclosing_volume` with the corresponding `GFile` as the first argument. In the finish callback passed to the operation, the file can then be accessed again. In Thunar, this process of mounting volumes before accessing their contents is being used in the side panel when a user clicks on of the not yet mounted volumes, which are distinguished from other volumes by their slightly transparent icons (figure 2.2).

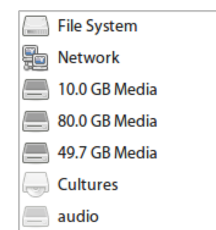

Figure 2.2. Volumes in the Thunar side panel.

Resolving Shortcuts

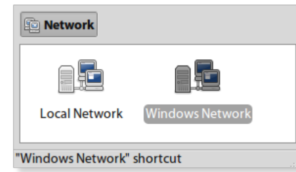

Figure 2.3. Windows workgroup shortcuts.

In GIO, shortcuts are virtual links to other files. They differ from symbolic links in that the link information is not stored on disk and that they are only virtually pointing to another URI. They may also reference files across different URI schemes while symbolic links are restricted to files located on the same partition. The `GFileInfo` objects of all shortcuts have a `standard::target-uri` attribute. When a shortcut is to be opened, the URI referred to in this attribute has to be loaded instead. Shortcuts are being used by GVfs for Windows network shares and workgroups, as can be seen in figure 2.3.

Mounting and Resolving Mountable Files

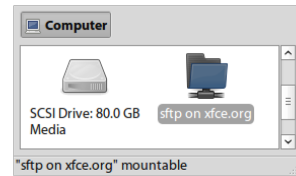

Figure 2.4. Drive and SFTP mountables in Thunar.

Mountable files are used to represent volumes and their mount points or root folders. They may point to mountable drives or partitions as well as virtual/remote locations such as SFTP hosts or FTP base URIs. They can be mounted asynchronously via `g_mountable_mount` which is very similar to `g_volume_mount`. Once mounted, the application is supposed to load and open the URI referred to in the `standard::target-uri` attribute. At the time of writing, mountables only appear in the `computer://` URI scheme in order to list drives and partitions currently connected and virtual locations currently mounted, as is shown in figure 2.4.

Unified Solution Implemented in Thunar

To reduce the amount of redundant and copied code in Thunar, a unified way to handle the situations described above was implemented. It might be of interest for other applications or libraries that aim at providing file browsing capabilities based on GIO.

All classes dealing with opening directories or launching files implement a common interface called `ThunarBrowser`. As in C++, which aside from virtual methods has no interface concept, GObject interfaces can have properties, signals and methods. The `ThunarBrowser` interface introduced along with the migration has two methods called `thunar_browser_poke_file` and `thunar_browser_poke_volume`, which are shown in listing 2.1. These methods are very similar to some of the APIs provided by GIO in that they are asynchronous and along with other parameters take a callback for when the

```
typedef void (*ThunarBrowserPokeFileFunc) (ThunarBrowser *browser,
                                           ThunarFile *file,
                                           ThunarFile *target_file,
                                           GError *error, gpointer user_data);

typedef void (*ThunarBrowserPokeVolumeFunc) (ThunarBrowser *browser,
                                             GVolume *volume,
                                             ThunarFile *mount_point,
                                             GError *error, gpointer user_data);

void thunar_browser_poke_file (ThunarBrowser *browser, ThunarFile *file,
                               gpointer widget, ThunarBrowserPokeFileFunc func,
                               gpointer user_data);

void thunar_browser_poke_volume (ThunarBrowser *browser, GVolume *volume,
                                 gpointer widget, ThunarBrowserPokeVolumeFunc func,
                                 gpointer user_data);
```

Listing 2.1. The `ThunarBrowser` methods for resolving files and volumes.

operation is finished. When a volume or a file needs to be resolved in one of the ways described above, these methods can be used and the mount point or target file can be processed in the callback. The `ThunarBrowser` class itself wraps 500 lines worth of special case treatment and and error handling for all of the scenarious presented in this section.

A very simple use case of this, taken from the location entry into which the user can freely enter arbitrary URIs and local paths, is shown in listing 2.2. The entry point is the function `thunar_location_entry_activate` which is called as soon as the user confirms something he entered by pressing the return key.

This example demonstrates how the resolving mechanism works. Normal files on volumes already mounted are directly forwarded to the finish callback. In this case the `file` and `target_file` parameters are equal. Files on volumes that have not been mounted yet, such as SFTP or FTP connections that are not yet established, first have their volumes mounted, which sometimes involves user interaction for account credentials. After that, the file is reloaded and passed to the finish callback with `file` and `target_file` pointing to the same object once again. Mountable files are first mounted asynchronously. Like with shortcuts their target URI is resolved into a `ThunarFile` object afterwards and then passed to the finish callback. This time, `file` points to the mountable file or the shortcut while `target_file` refers to the file into which it was resolved.

```
static void
thunar_location_entry_poke_file_finish (ThunarBrowser *browser,
                                        ThunarFile    *file,
                                        ThunarFile    *target_file,
                                        GError        *error,
                                        gpointer       ignored)
{
  _thunar_return_if_fail (THUNAR_IS_LOCATION_ENTRY (browser));
  _thunar_return_if_fail (THUNAR_IS_FILE (file));

  if (error == NULL)
    {
      /* try to open or launch the target file */
      thunar_location_entry_open_or_launch (THUNAR_LOCATION_ENTRY (browser),
                                            target_file);
    }
  else
    {
      /* display an error explaining why we couldn't open/mount the file */
      thunar_dialogs_show_error (THUNAR_LOCATION_ENTRY (browser)->path_entry,
                                 error, _("Failed to open \"%s\""),
                                 thunar_file_get_display_name (file));
    }
}

static void
thunar_location_entry_activate (GtkWidget           *path_entry,
                                ThunarLocationEntry *location_entry)
{
  ThunarFile *file;

  _thunar_return_if_fail (THUNAR_IS_LOCATION_ENTRY (location_entry));
  _thunar_return_if_fail (location_entry->path_entry == path_entry);

  /* determine the current file from the path entry */
  file = thunar_path_entry_get_current_file (THUNAR_PATH_ENTRY (path_entry));
  if (G_LIKELY (file != NULL))
    {
      thunar_browser_poke_file (THUNAR_BROWSER (location_entry), file, path_entry,
                                thunar_location_entry_poke_file_finish, NULL);
    }
}
```

Listing 2.2. Simple use case of the ThunarBrowser methods.

All in all, this has proven to be a very convenient API in all situations where the state of a file or volume due to the unpredictable nature of user interaction is not known beforehand.

2.4 Framework for Time-Consuming Operations

Many of the things a file manager does require multi-threading or at least asynchronous processing in order for the graphical user interface to remain responsive. When dealing with slow drives or files on other machines connected over the local network or the internet, this includes everything down to the very atomic operations. If, for instance, the network connection is interrupted or even lost, and the application waits for the result of a synchronous file information query, the usual network timeout is certainly not an acceptable response time for the user interface.

Refactoring an existing application that was not designed to be event-driven in all possible situations is challenge on its own. Thus, this section focuses on less atomic operations like copying files or listing the contents of a folder, which were already performed asynchronously in Thunar before the migration. In section 2.2 the major differences of the file operation APIs of GIO and ThunarVFS were discussed. It was concluded that the more generic high-level API of ThunarVFS is a much better approach, especially for recursive operations which are simply not present in GIO itself.

For the migration this means that the code for `ThunarVfsJob`, `ThunarVfsTransferJob`, and `ThunarVfsSimpleJob` either has to be integrated into the source code of Thunar itself, or into one of the libraries Thunar already depends on. Listing 2.3 shows that, aside from the `infos-ready` and `ask-replace` signals, `ThunarVfsJob` as well as `ThunarVfsSimpleJob` are not limited to to file operations at all. In fact, they can be seen as general-purpose classes to ease the implementation of *any* threaded or asynchronous operation. They provide basic features to create jobs based on a subclass of `ThunarVfsJob` or simply a delegate, which traditionally is a function pointer. Subclasses of `ThunarVfsJob` can override the `execute` method which is processed in a separate thread once `launch` has been called. Delegates are like the `execute` method and allow less complex jobs to be defined without subclassing.

Some of the signals, like `info-message` or `percent` or can be emitted from within the `execute` function or the delegate. Others, like `error` or `finished` are emitted by `ThunarVfsJob` depending on the success of the job. The signals `ask` and `ask-replace` can be used in case user interaction is needed. It should be noted that `ThunarVfsJob` takes care of synchronizing threads when emitting signals, which makes life much easier for developers.

```
enum ThunarVfsJobResponse {
  YES, YES_ALL, NO, NO_ALL, CANCEL, SKIP
}

class ThunarVfsJob : GLib.Object {
  ThunarVfsJob launch ();
  void cancel ();
  bool cancelled ();

  abstract void execute ();

  signal void error (GError error);
  signal void finished ();
  signal void percent (double percent);
  signal void info-message (string message);
  signal void infos-ready (List<ThunarVfsInfo> thunar_vfs_infos);
  signal ThunarVfsJobResponse ask (string message, ThunarVfsJobResponse choices);
  signal ThunarVfsJobResponse ask_replace (ThunarVfsInfo src_info,
                                          ThunarVfsInfo dst_info);
}

delegate bool ThunarVfsSimpleJobFunc (Job job, GValue[] values) throws GLib.Error;

class ThunarVfsSimpleJob : ThunarVfsJob {
  ThunarVfsJob launch (ThunarVfsSimpleJobFunc job_function,
                       unsigned int           n_values, ...);
}
```

Listing 2.3. Declarations of ThunarVfsJob and ThunarVfsSimpleJob in simplified Vala syntax.

Threads are rather difficult to work with, especially in a language like C which lacks built-in synchronization features. The ThunarVFS job classes provide a convenient and easy-to-use alternative to manual synchronization and low-level thread management. The demonstrated flexibility of ThunarVfsJob and ThunarVfsSimpleJob thus makes them good candidates for inclusion in a library called exo (the Xfce extension library) which is already being used by many Xfce core applications like Thunar, xfdesktop and xfce4-panel. By moving the two classes into this library, everything written for Xfce could benefit from them.

Unsurprisingly, this is what happened during the migration. The classes were not only renamed to ExoJob and ExoSimpleJob, they were also rewritten from scratch. Instead of synchronizing threads manually based on GThread, the threading subsystem of

GLib, as was done in `ThunarVfsJob`, `ExoJob` was implemented using `GIOScheduler`, a scheduler for asynchronous operations with integration into the GLib main event loop, mainly used inside GIO. Both, the `execute` function and the `ThunarVfsSimpleJobFunc` delegate also gained an additional `GCancellable` parameter to replace the boolean cancel flag previously being used.

The last remaining class, `ThunarVfsTransferJob`, and all convenience functions such as `thunar_vfs_copy_file` and `thunar_vfs_list_directory` were renamed and moved into Thunar itself, again not without being rewritten almost in their entirety. Naturally, all file operations were implemented on top of lower-level ThunarVFS functions before, and had to be migrated to GIO. Even though not quite as simple as a search and replace task, this is not covered here.

2.5 Thumbnail Generation Using the Tumbler D-Bus Service

Generating and managing preview images for files, also called thumbnails, turns out to be more complex than one would imagine. The thumbnail managing standard [2], a widely adopted freedesktop.org specification, defines how and where thumbnails are stored.

As discussed in section 1.2, GIO supports this standard in only one direction—loading thumbnail information. There are no functions for generating and storing thumbnails, most likely because the means by which thumbnails are generated usually vary between different applications. GNOME applications traditionally use `GnomeThumbnailFactory` from a library called libgnomeui and the image editor GIMP has its own built-in thumbnailing code, while Thunar has always used ThunarVFS for generating thumbnails.

Thumbnailing in ThunarVFS

ThunarVFS provides an implementation for generating thumbnails and saving them in compliance with the thumbnail managing standard [2] called `ThunarVfsThumbFactory`. Its design is very similar to `GnomeThumbnailFactory` in many regards. Its API design is shown in listing 2.4.

In addition to built-in thumbnailing code for several image formats based on GdkPixbuf, JPEG and fonts, ThunarVFS also allows for custom thumbnailers to be written. These are defined using desktop entries including information such as supported MIME types and a shell command, as demonstrated in listing 2.5. When being called from ThunarVFS, these thumbnailers write the generated thumbnail image into a temporary file which is then moved to its final location on the disk by ThunarVFS. This way cus-

tom thumbnailers only have to worry about generating the thumbnails, not about saving them in compliance with the standard which also involves avoiding race conditions during saving.

```
enum ThunarVfsThumbSize {
  NORMAL, LARGE
}

class ThunarVfsThumbFactory : GLib.Object {
  ThunarVfsThumbFactory (ThunarVfsThumbSize size);
  string lookup_thumbnail (ThunarVfsInfo info);
  bool can_thumbnail (ThunarVfsInfo info);
  bool has_failed_thumbnail (ThunarVfsInfo info);
  Gdk.Pixbuf generate_thumbnail (ThunarVfsInfo info);
  bool store_thumbnail (Gdk.Pixbuf pixbuf, ThunarVfsInfo info) throws GLib.Error;
}

string thunar_vfs_thumbnail_for_path (ThunarVfsPath path, ThunarVfsThumbSize size);
bool thunar_vfs_thumbnail_is_valid (string thumbnail, string uri,
                                    ThunarVfsFileTime mtime);
```

Listing 2.4. API of `ThunarVfsThumbFactory` in simplified Vala syntax.

```
[Desktop Entry]
Version=1.0
Encoding=UTF-8
Type=X-Thumbnailer
Name=Ogg Thumbnailer
TryExec=ffmpegthumbnailer
MimeType=application/ogg;
X-Thumbnailer-Exec=/usr/libexec/ogg-thumbnailer %i %o %s
```

Listing 2.5. Example of a custom ThunarVFS thumbnailer definition.

As a consequence of the migration away from ThunarVFS, the thumbnailing code faces the same situation as the job classes covered in the previous section. There are several alternatives for dealing with this. Like `ThunarVfsJob` and `ThunarVfsSimpleJob`, `ThunarVfsThumbFactory` could be moved into exo, and thereby made available to other applications like image viewers or desktop managers. The important of this is not to be underestimated which makes the second alternative, moving thumbnailing into the Thunar source code itself, less realistic.

A D-Bus Specification for Generating Thumbnails

An approach very popular nowadays is to implement features required by multiple applications as D-Bus services, thus making them globally available and loadable on demand. A specification called the thumbnail management D-Bus specification [4] was recently drafted in an attempt to standardize the way thumbnails are generated across all desktop environments and applications. This specification not only defines the public interface for separate D-Bus services for generating and managing thumbnails as well as querying supported MIME types. It also specifies how custom (specialized) thumbnailer D-Bus services have to be installed in order to be recognized by the main services defined in the specification. This draft has been adopted by Nokia for their Hildon application framework which was the foundation of the Maemo platform for mobile devices until Nokia

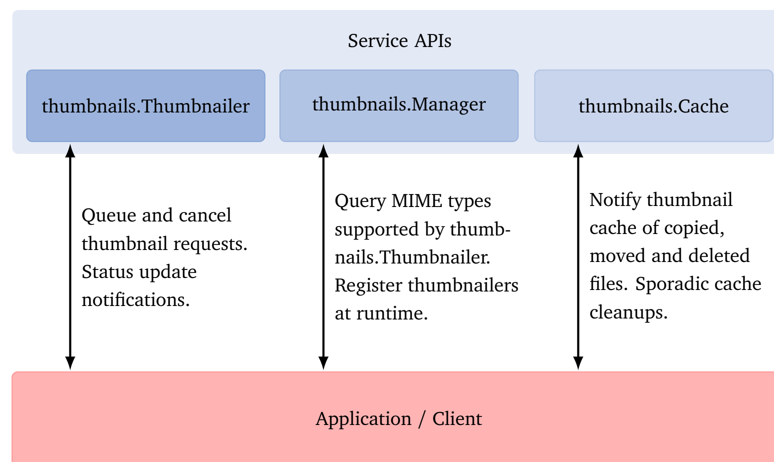

Figure 2.5. Service architecture defined by the thumbnail management D-Bus specification

announced the move from GTK+ to Qt, another GUI framework popular for powering the KDE desktop environment, in July 2009. The basic service architecture defined by the specification is shown in figure 2.5.

An investigation of the details of the specification including discussions with its authors revealed that the specified services would generally be powerful enough to serve as a replacement for the ThunarVFS thumbnailing code. Its most important D-Bus interface, `org.freedesktop.thumbnails.Thumbnailer`, shown in listing 2.6, provides methods for queuing (`Queue`) and cancelling (`Unqueue`) thumbnail requests as well as signals for when a request is started, finished or some URIs cannot be processed (`Error`). There is also a signal to notify clients of successfully created thumbnails (`Ready`), thus leaving the decision how many `Ready` signals per request are emitted up to the service implemen-

```
<?xml version="1.0" encoding="UTF-8"?>
<node name="/org/freedesktop/thumbnails/Thumbnailer">
  <interface name="org.freedesktop.thumbnails.Thumbnailer">
    <method name="Queue">
      <annotation name="org.freedesktop.DBus.GLib.Async" value="true"/>
      <arg type="as" name="uris" direction="in" />
      <arg type="as" name="mime_hints" direction="in" />
      <arg type="u" name="handle_to_unqueue" direction="in" />
      <arg type="u" name="handle" direction="out" />
    </method>
    <method name="Unqueue">
      <annotation name="org.freedesktop.DBus.GLib.Async" value="true"/>
      <arg type="u" name="handle" direction="in" />
    </method>
    <signal name="Started">
      <arg type="u" name="handle" />
    </signal>
    <signal name="Finished">
      <arg type="u" name="handle" />
    </signal>
    <signal name="Ready">
      <arg type="as" name="uris" />
    </signal>
    <signal name="Error">
      <arg type="u" name="handle" />
      <arg type="as" name="failed_uris" />
      <arg type="i" name="error_code" />
      <arg type="s" name="message" />
    </signal>
  </interface>
</node>
```

Listing 2.6. D-Bus interface definition for the thumbnailer service.

tation. Overall, this API is even more flexible than `ThunarVfsThumbFactory`, as it allows batch processing of thumbnails as an alternative to requesting thumbnails for individual files.

In the beginning, the specification was written for album artwork and similar features on mobile devices. As a file manager, its Thunar of course has very different requirements. High throughput is essential for browsing directories with a lot of files. Being an inter-process communication system, D-Bus has certain performance drawbacks as messages sent between applications obviously are not delivered as quickly as data passed between threads. Thus, the decision wether or not the thumbnail management D-Bus specification is applicable in the context of a file manager can only be made based on testing. Notable delays and system lags would speak against its adoption.

Fortunately, a centralized D-Bus service also has certain advantages. The use of D-Bus generally aids in establishing a loose coupling of applications and libraries. With a standardized D-Bus thumbnailer interface, implementations can be replaced without the applications noticing. Memory usage can be reduced as information about custom (or specialized) thumbnailer scripts are only stored in memory once. A centralized service also allows for fine-tuning and the employment of clever scheduling algorithms for a better overall performance.

Implementing the D-Bus Specification for Thunar

Along with the migration of Thunar to GIO, an implementation of this specification, called Tumbler, was implemented and later integrated into Thunar. The name, taken from the tank-like Batmobile appearing in Christopher Nolan's Batman movies, coincidentally derives all its seven letters from the word "thumbnailer" in exactly the same order they appear in it. Like Thunar, it is written in C based on GObject. Inspired by Hildon's hildon-thumbnail, it takes the concept one step further by being designed in a more object-oriented fashion, supporting different schedulers as well as plugins in addition to the specialized D-Bus thumbnailers defined in the specification. It also adds an abstraction layer on top of the thumbnail cache so that in theory, thumbnails can be stored deviating from the thumbnail managing standard [2] without affecting applications using the service. Different thumbnail caches and specialized thumbnailers can be implemented as plugins, the latter helping in drastically reducing the amount of traffic sent over D-Bus. For third-party and proprietary thumbnailers the preferred way still is to be implemented as D-Bus services in adherence to the specification.

Migrating Thunar to Tumbler for thumbnail generation can be seen as a two-step process. One part of it is to replace references to `ThunarVfsThumbFactory` in the Thunar

source code with more or less equivalent Tumbler client code. The other part is to migrate existing ThunarVFS thumbnailers to Tumbler, which includes writing a Tumbler plugin capable of loading custom ThunarVFS thumbnailers from files like the one shown in listing 2.5. Only the first part is covered here, as it provides a general insight into how to implement the client-side part of the thumbnail management D-Bus specification in an application.

Implementing a Thumbnailer Client

A very simple thumbnailer client is shown in listing 2.7. Implementing such a client is done by establishing a connection to the D-Bus session bus, creating an object for the D-Bus interface and connecting to its signals. After these initial steps, methods on this object can be called synchronously or asynchronously.

The example client shown in listing 2.7 first connects to the bus, and creates an object for the `org.freedesktop.thumbnails.Thumbnailer` interface. It then connects to the `Started`, `Finished`, `Ready` and `Error` signals which are defined in the thumbnail management D-Bus specification. When started, it queues thumbnail requests for two images, one with a valid and the other with an invalid MIME type. The basic communication with the thumbnailer service is demonstrated in listing 2.8 which shows the output of the example client. For each request, the `Started` signal is emitted first, followed by either a `Ready` or an `Error` signal. Each request is finished off with a `Finished` signal. If the ability to cancel requests and to query MIME types supported by the service before requesting thumbnails is not important, then this is all a thumbnailer client needs to do.

```
request queued: handle=85
request queued: handle=86
started/finished (handle=85)
ready (uris[0]=file:///tmp/0016.png)
started/finished (handle=85)
started/finished (handle=86)
error (handle=86, failed_uris[0]=file:///tmp/0016.png, code=1,
  msg=No thumbnailer available for 'file:///tmp/0016.png')
started/finished (handle=86)
```

Listing 2.8. Output of the example thumbnailer client.

When performance and responsiveness become an issue, however, more advanced techniques have to be employed in the client-side implementation of the service API. During the migration to GIO, Thunar gained a new class called `ThunarThumbnailer` which

```
using GLib; using DBus;

public class TumblerClient : GLib.Object {
  private DBus.Connection connection; private dynamic DBus.Object proxy;

  private void started_or_finished (dynamic DBus.Object proxy, uint32 handle) {
    print ("started/finished (handle=%ld)\n", handle);
  }

  private void ready (dynamic DBus.Object proxy, string[] uris) {
    print ("ready (uris[0]=%s)\n", uris[0]);
  }

  private void error (dynamic DBus.Object proxy, uint32 handle,
                      string[] failed_uris, int32 code, string msg)
  {
    print ("error (handle=%ld, failed_uris[0]=%s, code=%d, msg=%s)\n",
           handle, failed_uris[0], code, msg);
  }

  public void run () {
    connection = DBus.Bus.get (DBus.BusType.SESSION);
    proxy = connection.get_object ("org.freedesktop.thumbnails.Thumbnailer",
                                   "/org/freedesktop/thumbnails/Thumbnailer",
                                   "org.freedesktop.thumbnails.Thumbnailer");
    proxy.Started += started_or_finished; proxy.Finished += started_or_finished;
    proxy.Ready += ready; proxy.Error += error;

    uint32 handle;

    proxy.Queue(new string[] { "file:///tmp/0016.png" },
                new string[] { "image/png" }, (uint32) 0, out handle);
    print ("request queued: handle=%ld\n", handle);

    proxy.Queue(new string[] { "file:///tmp/0016.png" },
                new string[] { "invalid/mime-type" }, (uint32) 0, out handle);
    print ("request queued: handle=%ld\n", handle);
  }

  public static int main (string[] args) {
    var loop = new MainLoop (null, false); var client = new TumblerClient ();
    client.run (); loop.run (); return 0;
  }
}
```

Listing 2.7. Example thumbnailer client written in Vala.

uses several tricks to make thumbnail requests more efficient while interfering with the application's main loop as little as possible.

Thunar uses tree, list and icon views to display folders to the user. In GTK+, this involves a model/view concept with cell renderers for drawing the contents of individual cells. In Thunar, each view has a cell renderer specifically targeted at drawing thumbnails for the files being displayed. Once a file becomes visible in the view, it is piped through this renderer which then draws the thumbnail on the screen. This is the point where Thunar generates thumbnails for individual files on-demand. The fact that cell renderers are stateless (renderers don't own the objects they are drawing; a single cell renderer draws all the objects) generates one thumbnail request per file which is very inefficient.

`ThunarThumbnailer` is more or less used as a singleton class in Thunar, which solves the above issue by using a wait queue that is processed at most every 100 milliseconds. Individual thumbnail requests made in this time frame are grouped and sent out over D-Bus as a single request. Compared to sending individual requests which for n thumbnails generates $4n$ D-Bus messages (`Queue` call and n `Started`, `Ready`/`Error` and `Finished` signals), this approach can reduce the D-Bus overhead to $3 + n$ messages (one `Queue` call, one `Started` and `Finished` signal, n `Ready`/`Error` signals). This calculation only holds in static situations with no user interaction. However, the more individual thumbnail requests are made within the 100 millisecond time frame, the better this technique performs. This is particularly important when a user scrolls up and down obsessively.

In order to avoid pointless requests, files are filtered before they enter the wait queue. The `org.freedesktop.thumbnails.Manager` service is used to query a list of supported MIME types which is kept in sync over the runtime of the application. Like files that are already in the wait queue and files that are already being processed by the thumbnaile rservice, files with unsupported types are not added to the wait queue and are simply ignored.

To stay out of the way of the application's main loop, all D-Bus calls are performed asynchronously. As the application does not receive request handles and other replies immediately, `ThunarThumbnailer` needs to manage internal request identifiers in addition to the request handles returned by the thumbnailer service. Two hash tables are employed to establish a bidirectional mapping between them. The class also remembers which URIs are associated with each request. These mappings are then used in the signal handlers. All signal handlers utilize idle functions, which means that the file objects corresponding to a request are only updated when there are no other events pending in the main loop. All this allows for lag-free scrolling in large directories while thumbnails are generated in the background.

Whether the `ThunarThumbnailer` class will moved into exo in the future is unclear. However, considering the complexity it hides from applications it might be worth making it available to others. Image viewers with thumbnail bars have similar requirements and would thus benefit from it.

3 Summary and Evaluation

The evaluation sums up what was implemented and goes on to describe what benefit and implications this has and whether there were any regressions introduced. On August 21st, 2009, all changes made during the migration were merged into the main development branch of Thunar, which will eventually lead to its next stable release. Besides dropping ThunarVFS by replacing it with GIO, the migration also introduces new features visible to the end user, some of which are described in section **??**. Quite naturally, the migration was not carried out without regressions. The previous chapter already covered some of the API and functionality differences between ThunarVFS and GIO. As virtual file system parts of these two libraries (it was left out on purpose until now that GIO also includes APIs for network I/O) cover about the same functionality, the number of major regressions is small. Section 3.3 explains some of the issues that emerged due to the migration to GIO and need to be dealt with in the future.

3.1 Summary of the Changes

Overall, 271 files were changed during the migration, making for a total of 25,437 insertions and 42,458 deletions. Much of this is due to the removal of ThunarVFS, which alone includes 31,167 deletions. Directly added during the migration were 15,650 lines while 5,842 lines were deleted. All in all, 118 files were manually edited to realize the implementation of GIO in Thunar.

Much of the lines added to Thunar were part of reimplementing the file operation jobs from ThunarVFS based on GIO and moving them into Thunar itself. Almost as much effort was put into the other main parts of the migration: the thumbnailer D-Bus client implementation in Thunar and the code modifications related to loading file information as well as mounting and opening volumes and files asynchronously.

As described earlier, the base classes of the job framework was moved into exo, something other applications will be able to benefit from in the future. A program called xfce4-screenshooter, which takes screenshots of the desktop and allows users to save or upload them to the web has already adopted it for its implementation of the upload feature.

With Tumbler, a new D-Bus service for creating thumbnails desktop-wide, all in all worth around 7,000 physical lines of code spread across 50 source files, was designed and implemented during the migration. Not only is it extensible to support more file types, it also leaves room for optimization by making different scheduling algorithms for thumbnail requests possible. It is also not limited to Thunar but can be used by any desktop application, examples including image viewers and desktop managers. As it depends only on GLib and D-Bus, it might furthermore be an attractive solution for other environments. A future collaboration with the Maemo project has briefly been discussed.

3.2 New Features

An undertaking like this would not be half as interesting if there were no new features to be expected from it. The migration from ThunarVFS to GIO was mostly a technical one that was started with the goal to reduce the maintenance overhead and to reuse a well-tested piece of software that was already being used indirectly as a part of the software stack Thunar depends on. As such, most of the changes took place under the hood and most of the new features visible to the user are more of a consequence of moving towards GIO rather than being implemented as a part of the migration itself.

One of the key features of GIO is its extensibility. As was already elaborated on in the introduction chapter, it allows for a reuse of available GIO extensions, the most important being virtual file system implementations, in Thunar and other applications written on top of GIO. With GVfs in particular, a number of virtual and remote file systems are available and supported by Thunar out of the box. Their seemingless integration is guaranteed by a uniform GIO API and a number of modifications made to Thunar during the migration, such as the aforementioned asynchronous resolving and opening of volumes and files.

Thanks to GVfs and its virtual file system implementations for Windows network shares, SFTP, FTP and WebDAV, Thunar now integrates much better in private as well as corporate networks. These remote file systems have always been one of the features most desired by Thunar's user base. So much missed even that some Xfce-based distributions decided to ship their own patches and workarounds to enable rudimentary support for file sharing or remote file browsing. Where Thunar only supported local paths and `trash://` or `file://` URIs to be entered/opened by the user, it now supports all of the URI schemes supported by available GIO extensions in addition to local paths.

URIs can be mounted and opened by typing them into the location bar or dialog or by issuing commands like `thunar ssh://xfce.org/home/jannis` in a terminal. In addition

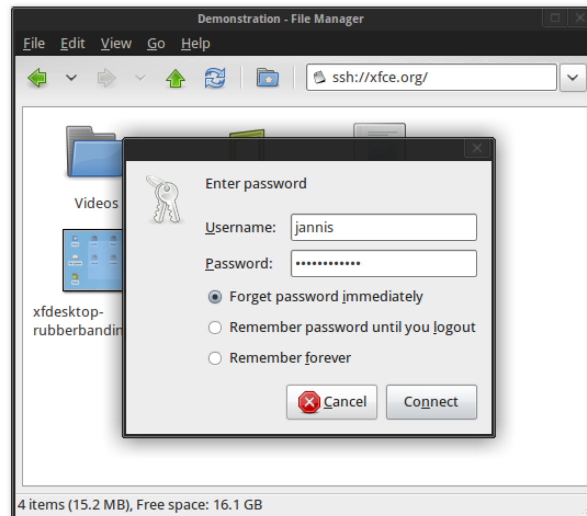

Figure 3.1. Thunar asking for SFTP username and password.

to this a network icon was added to the side panel in order to give users Windows environments faster access to files shared across the network. When opening a URI that requires login information, the user will be asked to enter his credentials, as can be seen in figure 3.1, before Thunar continues with mounting and opening the directory or file.

Not only file systems on remote machines are supported. GVfs also ships special implementations for browsing the contents of digital cameras, discovering available Bluetooth devices or viewing all available and already mounted volumes. While some of these involve cryptic URIs including hardware device identifiers, most of them are accessible in a similar way to the local file system.

All of these file systems are mounted as long as the active desktop session lasts. They are presented to the user like local folders and files. Files can be copied and moved around between all of these locations as long as they are writable. Even thumbnails are generated for all URI schemes supported by the Tumbler thumbnailers installed on the system. When the FUSE (Filesystem in USErspace) package is available, GVfs even provides a compatibility layer that mounts virtual file systems into the user's home directory and thus makes them accessible via local paths. This allows the file manager to hand remote files over to applications only supporting local paths.

The `trash://` scheme plays a special role in Thunar, as its availability has an influence on how delete operations are handled everywhere in the file manager. If it is supported files are by default moved to the trash when they are deleted. This is very convenient as they can be restored from there at a later point. If the trash implementa-

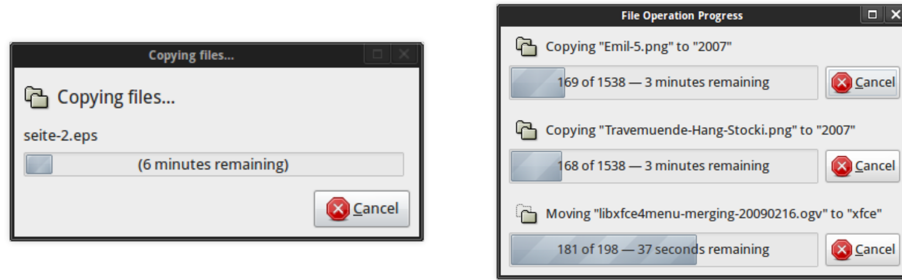

Figure 3.2. Shared progress dialog for file operations before and after.

tion is unavailable files cannot be trashed and are therefore deleted permanently once confirmed by the user. After earlier releases of Thunar, a lot of users have expressed an interest in disabling the trash functionality, something that has now become reality thanks to the extensibility of GIO.

Another user-visible feature that was implemented along with the migration and the refactoring of the job classes was a redesign and overall improvement of the file operation progress dialog. Thunar had traditionally used one progress dialog for each file operation. When the user copied files from one folder to another, he was presented with a dialog providing him with information about the progress of the operation and the time remaining. With each additional operation another progress dialog was opened.

People tend to do things in parallel. The desktop setup with multiple workspaces used on most systems except for Windows encourages to run applications for different tasks concurrently. It thus comes to no surprise that multiple progress dialogs for independent copy, move or delete operations are a very common scenario. One example would be image or movie folders where people choose the images or movies they want to copy to a USB stick one by one and launch copy operations as they go. With all these parallel tasks and operations opened or running, it is critical to keep an eye on the number of windows opened by an application. Avoiding clutter becomes an important aspect in designing an application.

Modern file browsers like Nautilus from GNOME or the OS X Finder today use a single dialog to visualize the progress of all file operations. During the migration, a similar shared progress dialog was implemented for Thunar. Figure 3.2 shows the new dialog in comparison to its predecessor which used more space for individual operations and was less descriptive by not mentioning the destination of copy and move operations or the source directory when deleting files. In addition to showing all active operations, the new dialog also shows a status icon in the notification area of the desktop panel, allowing the dialog to be hidden on demand.

Other Features

There are a few minor features that are not as noticable as the ones mentioned before. One of them is auto-completion not only for local files entered in the location bar or dialog of Thunar, but also for URIs on mounted remote file systems. In addition to that the Thunarx plugin interface has been updated so that plugins now have access to the `GFile` and `GFileInfo` objects associated with files and thus all their functionality is also available on virtual file systems.

3.3 Known Major Regressions

Luckily, the number of known major regressions after the migration is small. One that was already described briefly in section 2.2 is related to the `GAppInfo` interface of GIO. Gompared to similar classes from ThunarVFS, `GAppInfo` is limited. Not only does it no longer support so-called desktop actions, a now deprecated part of the desktop entry specification [1], which allowed applications to install context menu actions and its own application information in a single file. It also provides no information about whether an application can handle multiple URI or path arguments at once. The absence of this information can lead to problems when trying to open multiple selected files with an application via the context menu.

Another known regression is related to the dependencies of Thunar. Before the migration, HAL was the only external library required for volume management to work. Trashing files instead of deleting them permanently was supported out of the box without additional dependencies. At the current state after the migration neither volume management nor the trash are supported without dependencies which were not required before. GVfs has to be installed for both features to work, and, depending on the version of GVfs, either HAL or DeviceKit-disks together with gnome-disk-utility are required for volume management. That not being enough, GVfs also depends on a tool called gnome-mount for mounting removable drives and partitions which is hard-coded in the GVfs source code. Two additional dependencies enabled on most distributions are the GNOME configuration system GConf and gnome-keyring, which is used to manage the user's credentials entered when mounting remote file systems. While all these dependencies are reasonable and fulfill a clear purpose, they were not required before; a situation that has to be improved significantly considering that one of the main design goals of Xfce and also Thunar is to go easy on resources and thus, not to depend on libraries that can be avoided at little maintenance cost.

3.4 Consequences for Other Applications

The migration has several implications on other applications in the Xfce desktop environment. A lot of them, like ristretto, an image viewer, or xfburn, the Xfce burn software, or xfdesktop, the Xfce desktop manager, depend on ThunarVFS for file browsing features. The latter not only uses ThunarVFS for file icons on the desktop, it also hooks into Thunarx to query context menu items from Thunar plugins and relies on Thunar's D-Bus interface for opening folders and displaying the application chooser dialog when opening one or more files.

The 4.8 release of Xfce which is scheduled for April 2010 is intended to be the first release without ThunarVFS. As a consequence, all applications depending on it will have to be updated to use GIO instead. While this is not problematic for the small number of Xfce core components, there are also a number of applications and panel plugins written by other people outside the core Xfce development team. In order to keep those in a working state until their maintainers have time to migrate them, ThunarVFS has been moved into its own repository. It will most likely see will see one or two new releases until its development will be completely halted. This is done so that distributions can package the standalone version of ThunarVFS and update other packages which have not been migrated yet.

A new problem has emerged in the exo library at the very end of the migration. One of its command line tool, called exo-open, which is frequently used for opening arbitrary paths or URIs, and which decides whether to open these locations in Thunar, the web browser or the default email client now needs to handle a variable amount of additional URI schemes such as `network://` or `gphoto2://` (a URI scheme used for browsing the contents of digital cameras). It can no longer assume a fixed set of URI schemes (previously the `file://` and `trash://` schemes were hard-coded into ThunarVFS) and instead needs to determine which schemes are supported by the GIO extensions available, and which to open in the file manager. This directly or indirectly affects a lot of applications and custom scripts written by users.

An solution for this was recently implemented by Nick Schermer, another Xfce developer who at the time of writing is responsible for maintaining the exo library. The implementation extends GIO by creating default `GAppInfo` objects for various URI schemes. These are indirectly used when the exo-open utility calls the `gtk_show_uri` function from GTK+ to open a URI in an appropriate application. This method also ensures that URIs displayed in about dialogs and other GTK+ widgets are launched exactly the same way URIs are opened from the command line or user scripts.

4 Outlook

Even though the changes made during the migration have all been merged into Thunar, exo or into a separate package like Tumbler, this does not mean that the work ends here. The migration can be seen as a successful attempt to revive the development activity around Thunar which since the last release actively being worked on by Benedikt Meurer was more or less limited to bugfixes and minor improvements. The move towards GIO introduces new challenges, as some of the new features leave room for improvements in their integration into Thunar. But before this can be worked on, the changes have to be tested extensively, regressions have to be fixed and ideally, the impact of the migration to other applications such as the xfdesktop desktop manager has to be reduced. As a consequence, several key areas that need improvement stand out against the rest.

For a continued backwards-compatible support of custom ThunarVFS thumbnailer scripts, a Tumbler plugin is planned to be written which exposes the combined information about all thumbnailer scripts to Tumbler and acts as a proxy between Tumbler and the thumbnailer scripts by forwarding all work to the scripts when receiving thumbnail requests. Some of the built-in thumbnailers shipped with ThunarVFS also need to be rewritten or ported to Tumbler.

In order to avoid GVfs as a dependency in Thunar for volume management, writing a custom GIO volume monitor extension is currently under consideration, either based on HAL or DeviceKit-disks. Considering its ongoing deprecation, HAL might seem like a bad choice but unlike DeviceKit-disks it is already available on Unixes other than Linux. Portions from `ThunarVfsVolumeManager` based on HAL could probably be reused, which could make HAL a quick solution for Thunar 1.2 which is supposed to be released along with Xfce 4.8 in April 2010. A new implementation based on DeviceKit-disks could then be developed afterwards. On the other hand quite a number of bugs have been filed against the code based on HAL, and especially mounting NTFS partitions seems to be problematic. This is still being discussed and a solution will hopefully be available as part of the next Thunar release.

As explained in detail in sections 2.2 and 2.4, the file operation APIs in GIO are less powerful than than what ThunarVFS provides. Consequently, other Xfce applications, especially xfdesktop, will hit the same problems with complex, possibly recursive operations which were already solved in Thunar. Together with the additional work put into

the shared file operation progress dialog, this makes Thunar a good candidate for an extended D-Bus service which not only provides an interface to open or launch files and directories but also makes common dialogs and operations available to other parts of the desktop. As such, D-Bus methods for displaying file properties dialogs as well as for copying, moving or deleting files are planned to be added with next release of Thunar.

So far, the integration of various virtual and remote file systems supported by GIO via GVfs is only implemented at a very basic level. Directories and files on these file systems can be browsed or opened as if they were on stored locally. However, entry points have to be entered by the user manually. For some backends this requires a deeper technical knowledge and thus cannot live up to the user interface quality of other parts of Thunar. A better integration of these file systems is desirable. Gigolo, an application for managing and opening virtual file system bookmarks based on GIO and GVfs, is an example of how the access to such locations can be realized with a graphical user interface. Some ideas can probably be taken from there and integrated directly into Thunar.

Being able to manage Windows network shares is a key feature to make Thunar more interesting and useful especially in corporate networks. As GIO does not provide the required functionality to share folders over the network, other ways to achieve this will have to be examined. A plugin called thunar-shares-plugin, written by Daniel Morales, allows people to enable folder sharing in the folder properties dialog. It is, however, not possible to prepare the system for this without superuser access. Whether this is possible or not is left open for investigation.

Overall, the migration has been a success without any critical issues left to be resolved. The Thunar development has been revitalized and even though some of the areas that could be improved are nothing new (Windows network shares were never fully supported by Thunar itself), several opportunities for future enhancements have been uncovered. The migration to GIO provides a good starting point to making Thunar more powerful while maintaining its simplicity and the minimalistic user interface concept which has made it popular.

Literature

[1] Preston Brown, Jonathan Blandford, Owen Taylor, Vincent Untz, and Waldo Bastian. Desktop Entry Specification. http://standards.freedesktop.org/desktop-entry-spec/1.1/, 2004–2008.

[2] Jens Finke and Olivier Sessink. Thumbnail Managing Standard. http://jens.triq.net/thumbnail-spec/, 2001–2004.

[3] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part two: Media types. http://tools.ietf.org/html/rfc2046, 1996.

[4] Philip Van Hoof and Rob Taylor. Thumbnail management D-Bus specification. http://live.gnome.org/ThumbnailerSpec, 2009.

[5] Alexander Larsson. Plans for gnome-vfs replacement. http://mail.gnome.org/archives/gtk-devel-list/2006-September/msg00072.html, 2006.

[6] Thomas Leonard. Shared MIME-info Database. http://standards.freedesktop.org/shared-mime-info-spec/0.18/, 2004–2008.