GVFS: The making of a virtual filesystem

Alexander Larsson – alexl@redhat.com

Overview

- What is GVFS?
- Why do we need it?
- How does it work?
- How do I use it?

Why GVFS?

- Gnome-vfs not good enough
- Easy to use, modern API
- VFS at the right level in the stack
- Allow use of the vfs in Gtk+ (fileselector)
- Few dependencies, less bloat
- Project Ridley

Why is nobody using gnome-vfs?

- It has to many dependencies
- Wrong place in the stack
- API style from 1999 (not GObject based)
- Hard to use (URIs, xfer, no utility functions)
- Quality of implementation of some backends
- Design errors

Gnome-vfs design errors: posix model

- Not abstracted enough from posix model
  - Uses uid, gid and modes for permissions
  - Move vs check_same_fs
  - Supports read-write files
- Doesn't have some required highlevel features
  - Atomic save (with backups)
  - Fixed list of file info (modeled on struct stat)

Gnome-vfs design errors: Stateless I/O model

- Only persistent state is URIs and file handles
- Most backends need state (a connection)
  - Implicit state
- Authentication
  - Any operation can require authentication
  - Password dialogs show up at wrong times
  - Authentication doesn't know the context
  - Callbacks cause reentrancy
  - Gdk lock deadlock

Gnome-vfs design errors: In-process backends

- No way to share data between applications
  - Authentication
  - Connections
  - Caches
- Unstable backend can crash the application
- Backends have no control of their environment
- libraries used by backend linked into application

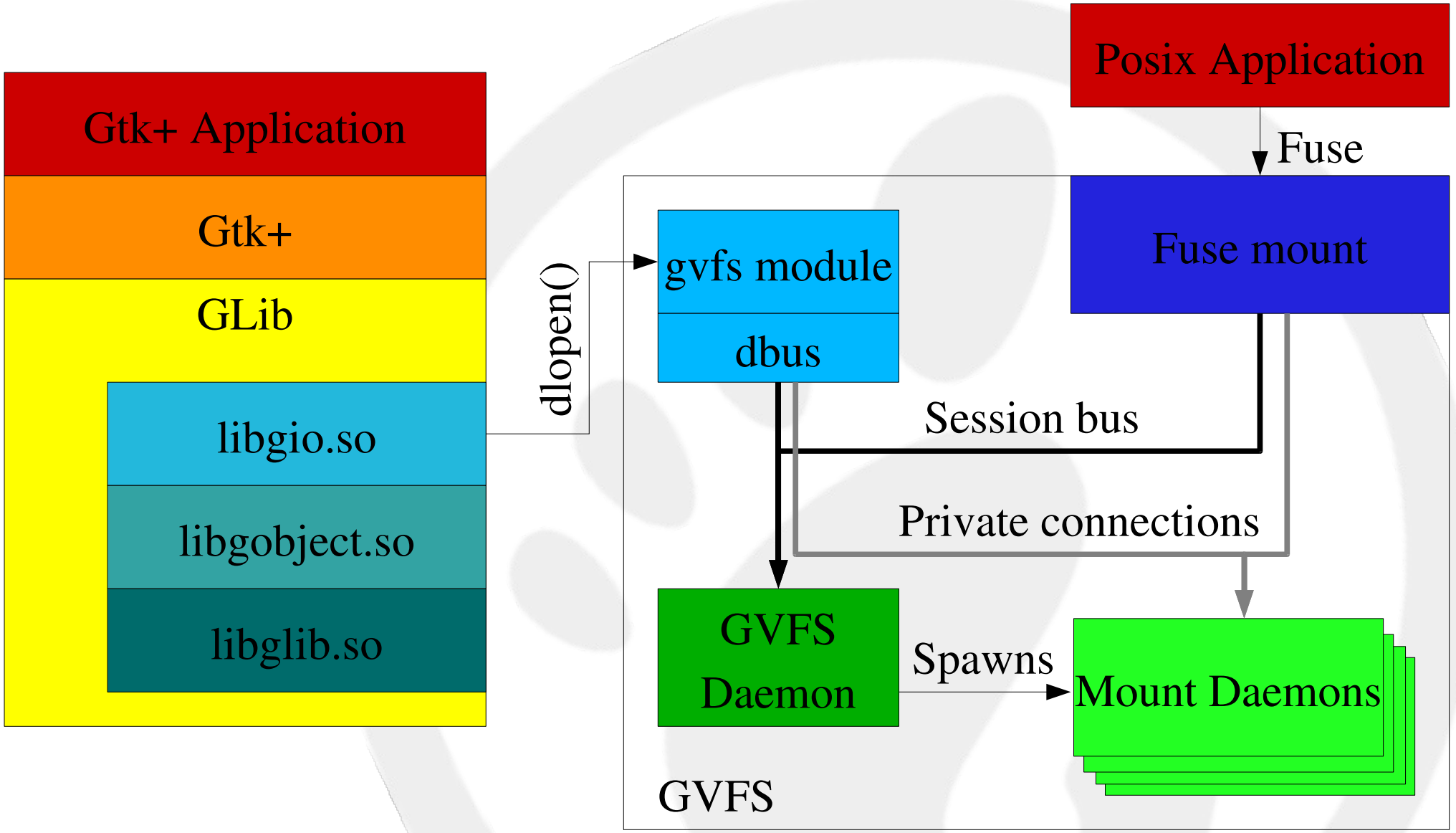
Gnome-vfs design errors: others

- Enforces threads
- Cancellation API has an unfixable race condition
- No display names or filename encoding support
- No icon support

How can we solve these issues

- Make API so good that developers prefer it
- Put it in glib
- Use GObject and other modern tools
- Make backends shared and out-of-process
- Only link minimal code into applications
- Make API more abstract
  - Less posix, more generic
  - Add highlevel features
- Make I/O statefull

High level architecture



libgio

- New library in glib, like gobject & gthread
- Uses gobject
- Provides a set of abstract APIs for I/O and files
  - Interfaces
  - Base classes
- Provides some implementations of these APIs
- Allows higher level components to interact even thought they are loosely coupled

gio – file system model

- Abstract file system model
  - GFile – reference to a file
  - GFileInfo – information about a file or filesystem
  - GFileEnumerator – list files in directories
  - Streams – read and write data

Posix	gio
`char *path`	`GFile *file`
`struct stat *buf`	`GFileInfo *info`
`struct statvfs *buf`	`GFileInfo *info`
`int fd`	`GInputStream *in, GOutputStream *out`
`DIR *`	`GFileEnumerator *enum`

gio – GFile

- GFile – a reference to a file
  - Construction:
    - `GFile *g_file_get_for_path (const char *path)`
    - `GFile *g_file_get_for_uri (const char *uri)`
    - `GFile *g_file_parse_name (const char *parse_name)`
    - `GFile *g_file_get_for_commandline_arg (const char *arg)`
    - Cheap, doesn't do I/O
    - Doesn't have to exist on disk
  - Can traverse the filesystem:
    - `GFile *g_file_get_parent (GFile *file)`
    - `GFile *g_file_get_child (GFile *file, const char *name)`
    - `GFile *g_file_resolve_relative (GFile *file, const char *relative_path);`
    - Never do path or URI string munging again!

gio – GFile operations

- g_file_enumerate_children -> GFileEnumerator
- g_file_get_info -> GFileInfo
- g_file_get_filesystem_info -> GFileInfo
- g_file_set_display_name
- g_file_set_attribute
- Other operations:
  - copy, move, delete, trash
  - make_directory, make_symbolic_link
  - mount, eject

gio – file data

- Reading files

```
GFileInputStream * g_file_read       (GFile          *file,
                                      GCancellable   *cancellable,
                                      GError        **error);
```

- Writing files

```
GFileOutputStream *g_file_append_to (GFile          *file,
                                     GCancellable   *cancellable,
                                     GError        **error);
GFileOutputStream *g_file_create    (GFile          *file,
                                     GCancellable   *cancellable,
                                     GError        **error);
GFileOutputStream *g_file_replace   (GFile          *file,
                                     time_t          mtime,
                                     gboolean        make_backup,
                                     GCancellable   *cancellable,
                                     GError        **error);
```

gio – streams

- Generic abstraction for I/O on streams of bytes
- `GInputStream`: read(), skip(), close()
- `GOutputStream`: write(), flush(), close()
- Optionally implements `GSeekable` interface
- Sharing stream APIs low in the stack allows loose coupling of components, eg:
  - GdkPixbuf reads from a `GInputStream`
  - Some library can read pixmap data from the net as a `GInputStream`
  - now GdkPixbuf can read the data from the net

gio – types of streams

- GInputStream – basic input stream
  - GFileInputStream – adds seek and get_file_info
  - GFilterInputStream – base class for wrapper streams
    - **GBufferedInputStream**
  - **GMemoryInputStream**
  - **GSocketInputStream**
- GOutputStream – basic output stream
  - GFileOutputStream – adds seek, truncate and get_file_info
  - GFilterOutputStream – base class for wrapper streams
    - **GBufferedOutputStream**
  - **GMemoryOutputStream**
  - **GSocketOutputStream**
- *GSeekable* – Interface that streams can support

gio – file information

- GFileInfo
  - set of key-value pairs
  - keys are namespaced strings “ns:key”
    - “std:name”, “std:size”, “unix:uid”, “access:read”
  - values are typed data
    - string, byte string, [u]int32, [u]int64, bool, object
- Getting a subset of attributes
  - `GFileAttributeMatcher`
  - `list of attributes, with wildcards`
  - “*”, “std:*”, “std:name,std:display_name”

gio – common GFileInfo namespaces

- std – type, names, size, content type, etc
- unix – unix specific info (inode, uid, etc)
- access – access rights
- mountable – can the location be mounted, etc
- time – access time, etc (not mtime)
- owner – string version of owner/group
- selinux – selinux info
- xattr – extended attributes
- extendable...

gio – file monitoring

- GFileMonitor (g_file_monitor_file)
- GDirectoryMonitor (g_file_monitor_directory)
- “changed” signal

```
void (* changed) (GFileMonitor* monitor,
                  GFile* file,
                  GFile* other_file,
                  GFileMonitorEvent event_type)
```

- types: changed, deleted, created, moved, attribute_changed, unmounted
- Backends: fam/gamin, inotify

gio – Asynchronous I/O

- Most operations availble in async version
- Default implementation using thread pool
- Uses default glib mainloop
- Threadsafe (but callbacks are in main thread)
- All callbacks use the same callback prototype

gio – Asynchronous API

- Async operations are split into two calls
  - _async() - starts the operation, then calls callback when ready
  - _finish() - gets the result from the call (typically called in callback)

```
typedef void (*GAsyncReadyCallback) (GObject *source_object,
                                     GAsyncResult *res,
                                     gpointer user_data);
void       g_file_op_async  (GFile *file,
                             input_arguments args,
                             GAsyncReadyCallback callback,
                             gpointer user_data);
ReturnType g_file_op_finish (GFile *file,
                             GAsyncResult *res,
                             output_args *args,
                             GError **error);
```

gio – File types

- content type
  - ASCII string defining the type of a file
  - platform dependent
    - mimetype on unix (shared mime spec)
    - extension on win32
    - OSX: Uniform Type Identifiers?
  - equals, is_a, is_unknown, get_descriptions, get_icon, get_mimetype, can_be_executable
- content guessing
  - sniffing (on unix)
  - filename matching

File handlers

- GAppInfo – Represents an installed application
  - Launch with filenames or URIs
  - Information: Name, Description, Icon
  - Implemented as desktop file on Unix
- Several ways to get
  - all installed
  - all for content type
  - default for content type
- Might move to Gtk+

Icons

- Need to support icons
  - content types
  - applications
  - files
- Problematic
  - Can't use GdkPixbuf or GtkIconTheme
  - Several types of icons in use
    - themed icons
    - filenames
    - custom backend icons (e.g. http favicons)

Icons – Solution

- `GIcon` interface
  - abstract
  - cachable (equals() and hash())
- `GLoadableIcon` interface
  - Load icon data as a stream
- Several Implementations
  - `GThemedIcon`
  - `GFileIcon`
- Icon renderer at Gtk+ level

gvfs – How it works

- gvfs module adds support for non-file: uris
- I/O on such files is implemented by talking to daemons handling each mount
  - general operations and metadata I/O use DBus
  - file content I/O use custom binary protocol
- main gvfs daemon
  - Found and autostarted via session bus
  - Starts and manages the mount daemons
  - desktop style config files for each mountable

gvfs – Fuse

- Allows backward compatibility
  - Open and save files with non-gvfs applications
  - Possible for file selector and file manager to reverse map to gvfs location
- One mountpoint (e.g. ~/.gvfs)
- Current mounts listed under toplevel dir

Some code

- g_file_load_contents
- g_file_replace_contents

Status

- Code in git
  - http://www.gnome.org/~alexl/git/gvfs.git
- API usable, but not fully done or frozen
- fuse mounts work
- smb backend availible
- ftp backend under development
- Important work before freezing
  - Nautilus port
  - File selector support

QA

Any questions?