“Now if we could get a solution to the home directory dotfile hell!”[11]

Making Linux NFS-mounted home directories useful again.

Andrei Warkentin
VMware, Inc.
andreiw@vmware.com

Abstract

Unix environments have traditionally consisted of multi-user and diverse multi-computer configurations, backed by expensive network-attached storage. The recent growth and proliferation of desktop- and single machine- centric GUI environments, however, has made it very difficult to share a network-mounted home directory across multiple machines. This is particularly noticeable in the context of concurrent graphical logins or logins into systems with a different installed software base.The typical offenders are the “modern” bits of software such as desktop environments (e.g. GNOME), services (dbus, PulseAudio), and applications (Firefox), which all abuse dotfiles.

Frequent changes to configuration format prevents the same set of configuration files from being easily used across even close versions of the same software. And whereas dotfiles historically contained read-once configuration, they are now misused for runtime lock files and writeable configuration databases, with no effort to guarantee correctness across concurrent accesses and differently-versioned components. Running such software concurrently, across different machines with a network mounted home directory, results in corruption, data loss, misbehavior and deadlock, as the majority of configuration is system-, machine- and installation- specific, rather than user-specific.

This paper explores a simpler alternative to rewriting all existing broken software, namely, implementing separate host-specific profiles via filesystem redirection of dotfile accesses. Several approaches are discussed and the presented solution, the Host Profile File System, although Linux-centric, can be easily adapted to other similar environments such as OS X, Solaris and the BSDs.

1 Introduction

The title of this paper has been kindly borrowed from a BYU UUG email [11], that originally inspired me to find a solution.

While some systems prefer a centralized approach to storing user-specific program configuration settings, Unix-like systems typically keep user preferences under a number of hidden files and directories kept in the root of the home directory. These hidden files and directories are distinguished by a leading dot in their name, and are thus generally called *dotfiles*. The historical behavior is to store read-once configuration to avoid hard-coded choices, for example `.profile` stores instructions for customizing the shell session, `.emacs` stores EMACS editor settings, and so on. Perhaps due to Unix-like systems being used in extensively networked, remote access and shared storage environments, dotfiles were never meant to be used as a persistent database for runtime-modified preferences, or as a locking mechanism, since that would have compromised the ability to log in concurrently into several machines with the same account stored on the network. There was always some degree of inflexibility, where the same configuration file would not work exactly as expected across different versions of software[1], yet the read-only nature of these files and conservative changes to setting schema meant you could come up with a valid subset of settings for all machines and operating systems in use.

The recent rise of “user-friendly”, graphical user interface-driven and largely PC-centric applications, however, has resulted in a number of popular software packages which are incompatible with the typical university or corporate heterogeneous environment based around network-mounted home directories and network logins. For recent software, such as the GNOME

[1] E.g., `.emacs` or `.vimrc` for the famous editors.

Desktop Environment, there are no simple solutions to making them work in an environment where network mounted home directories are a possibility, largely due to ever evolving and fluid configuration formats. Software distribution-specific minutiae and breaking changes across seemingly compatible major revisions of software, results in an inability to share the same configuration files. In environments where the user can make some guarantees as to software versioning and configuration compatibility, concurrent network logins are largely impossible due to the storage of writeable preferences and runtime data within dotfiles. Given that programs such as web browsers, WYSIWYG editors and desktop environments also manipulate configuration from within the applications, treating their dotfiles as writable databases, you immediately run into write collisions and configuration corruption when running concurrent sessions on several machines. Frequently, such software attempts to protect itself by creating lock files (see Figure 1), which results in denial of service condition during concurrent logins. Finally, the trend towards providing services via remote procedure call mechanisms, seen in software such as IPC buses and audio daemons like Enlightened Sound Daemon or PulseAudio, has resulted in storing named pipes, sockets, authentication cookies, and other unspeakable things within their dotfiles, with largely predictable results.

As an example of all of these, logging in to a Red Hat Linux RHEL 5 and SUSE Linux SLES11 system, concurrently or not, will result in a corrupted desktop on both hosts. Both of these systems use some variation of GNOME 2. A typical file system layout for GNOME 2 configuration can be seen in Figure 2. These issues are not new nor are they the only ones. Configuration file collisions across different versions of software, compounded by fragility and expressive verbosity for default auto-generated settings[2] has also been a bane for upgrading such software and its configuration successfully.

2 Related Work

The problem space itself is not particularly novel. Roaming User Profiles and Folder Redirection are two similar technologies available to Microsoft Windows users, which specifically deal with networked logins,

[2] As seen with GNOME 2, moving into GNOME 3.

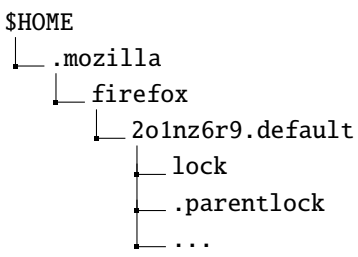

Figure 1: Partial structure of Mozilla Firefox configuration.

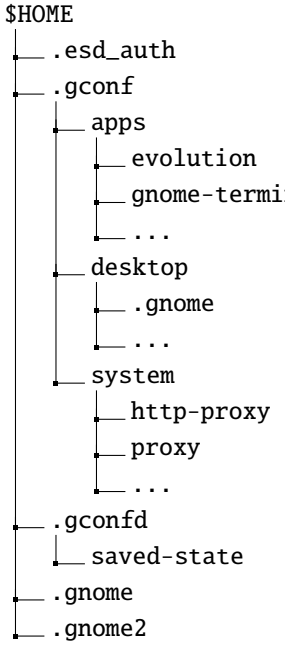

Figure 2: Typical structure of GNOME 2 configuration.

remote home directories and operating system-specific profiles [8].

Roaming User Profiles (RUP) synchronize the local copy of the user profile, consisting of user directories and a user-specific registry hive holding configuration entries, with the remote server upon login and logout. Conflicts are resolved based on modification time, and the registry hive is treated as an opaque binary object, with no fine grained synchronization. The weakest spot of the entire mechanism is specifically relying on a synchronization mechanism and being largely unaware of what is being synchronized. Copying data back and forth imposes a noticeable penalty, on the order of minutes, for anything but the most trivial profile, forcing users to store their data locally. By not relying on network file access and locking semantics for concurrent access, there is always the potential for silent data loss caused by the "last modified wins" policy or

by failures during synchronization. Additionally, RUP is not capable of distinguishing between synchronizing settings and synchronizing application data. A lack of fine-grained synchronization of registry keys, and a lack of state separation between user settings and host-specific user settings, results in inconsistent profile behavior across systems configured with a different set of applications or with different versions and revisions of Windows. These limitations are somewhat addressed by using completely separate profiles for Vista and newer versions, and by using the Folder Redirection mechanism to alias certain predefined user profile directories[3] to network locations, bridging the separate profiles to the degree that is possible and reducing the usability "threat" posed by synchronization.

A Roaming User Profile-like approach is not particularly feasible on Linux. Implementing such Windows semantics would mean using a local cache as the real home directory, which would then be synchronized under the user's credentials with the network copy on logging in and logging out. This relies on having a mechanism capable of resolving conflicts, and thus aware of all the possible applications, and having a complex conflict resolution policy. Getting this to work smoothly implies, at the very least, root access to all the machines affected, and some pretty serious source-level hacks[4] to get it all to work in a transparent and fail safe manner.

3 Solutions

Due to the limitations of a synchronization-based design, the proposed solution for the problems described above is dotfile access redirection. By redirecting dotfile accesses to local, host- or operating system- specific copies, we create separate configuration namespaces, thus resolving conflicts arising from concurrent accesses or versioning mismatches. A few approaches involving access redirection were investigated. The benefits of separate configuration namespaces are clear. Whatever the implementation details, a few design goals were kept in mind. In particular:

- All dotfile accesses are redirected.
- Accesses to dotfiles from each host are redirected to a special directory, specific for that environment.

[3] My Documents, etc.

[4] I, of course, mean changes to the authorization and authentication system, PAM.

```
$HOME
├── profiles
│   ├── andreiw-lnx ...  Dotfiles
│   │                    specific to host
│   │                    andreiw-lnx.
│   │   ├── .gnome2
│   │   ├── .xyzzy
│   │   └── ...
│   ├── andreiw-vmw ...  Dotfiles
│   │                    specific to host
│   │                    andreiw-vmw.
│   │   ├── .gnome2
│   │   └── ...
│   └── andreiw-vm1 ...  Dotfiles
│                        specific to host
│                        andreiw-vm1.
│       ├── .gnome2
│       └── ...
├── .gnome2
├── .xyzzy ...  Visible here because
│               we are on host
│               andreiw-lnx.
└── ...
```

Figure 3: View of a home directory on host andreiw-lnx, with dotfile redirection enabled.

- Mechanism and policy are separate.
- System configuration changes are minimal, ideally not requiring root access.

With this design there are, by default, no configuration collisions, conflicts and deadlocks. If a few of the dotfiles can indeed be safely shared, then a few strategic symbolic links can be employed. Learning from RUP, which suffers both from a poor mechanism and from mixing both mechanism and policy, the user has full control over how the redirection target directory is picked, whether it is by host name, IP or Ethernet address, or something completely different. The resulting mechanism is very flexible. See Figure 3 for a typical home directory layout with this design. Note that the visible dotfiles in the home directory root are really located inside the environment-specific directory.

One investigated approach was to override standard C library calls by loading a custom library with the `LD_PRELOAD` environment variable, similar to how the `fakeroot` package works [3]. The `LD_PRELOAD` environment variable signals the dynamic linker to load a specified library and attempt resolving symbols be-

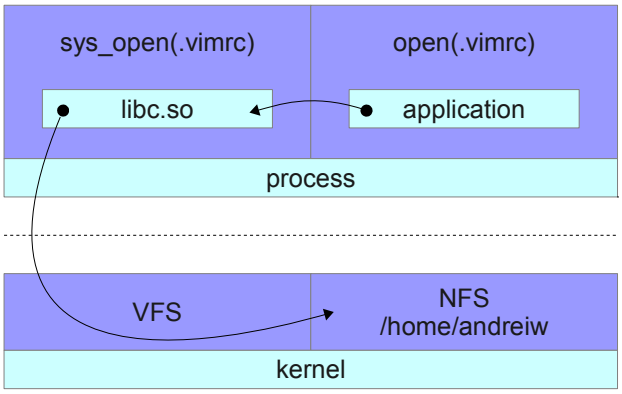

Figure 4: Regular `open()` path for an application.

fore loading all the libraries required by the loaded executable. Conceptually, calls like `open()`, `chmod()`, `unlink()` and the rest could be intercepted and modified to redirect accesses to dotfiles elsewhere (see Figures 4 and 5). In practice, however, this suffers from a few problems. The most significant issue is that it works only for executables dynamically linked to the C library. Statically linked software would bypass the redirection completely. Additionally, the method is very fragile and operating system dependent. The preloaded redirector would need to be tailored for the specific version of the C library in the system, as certain functionality is exposed and implemented differently, like the `stat()` and `mknod` family of routines[5]. The redirector would also need to properly handle both absolute and relative accesses to ensure isolation against malicious activity, and finally, would need to deal with the the `*at()`[6] variants, which operate relative to an opened file descriptor, by maintaining state about every opened file descriptor.

Another alternative implementation relied on `ptrace()`, coupled with process memory patching to intercept and redirect actual system calls, as `fakeroot-ng` does [3]. This solves the C library dependence issues along with being able to redirect calls made by statically-linked executables, yet at a cost of architecture dependence and severe performance penalties [5].

The long-term solution would be the FreeDesktop.org XDG Base Directory Specification [2], which separates

[5]GNU libc, for example, wraps `mknod()` with a versioned call to `xmknod()`, and `stat()` as versioned calls to `xstat()`, `fxstat()`, and `lxstat()`. See `/usr/include/stat.h`.

[6]`openat()`, `faccessat()` and similar calls were added in Linux 2.6.18, and are meant to address race conditions resulting from opening files in directories other than the current one [7].

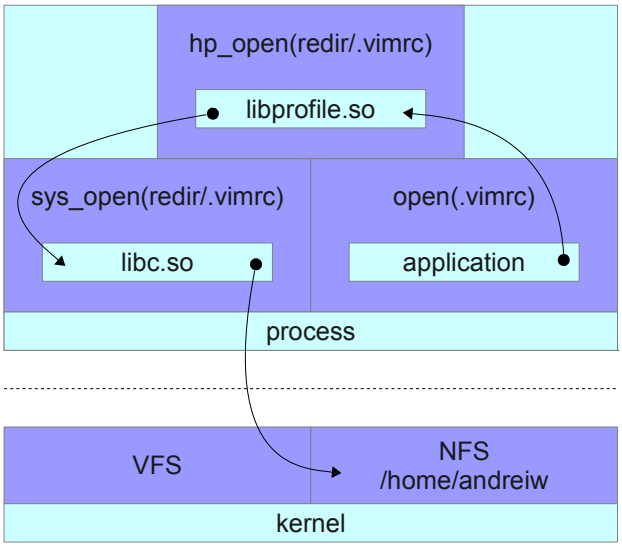

Figure 5: Modified `open()` path for an application with a file system redirector loaded using `LD_PRELOAD`.

application user data, caching-related, configuration and runtime-specific application data across four base directories, specified by environment variables as illustrated in Figure 6. In an environment where concurrent logins are expected, some or all of the base directories can be redirected to locations specific to the OS or host used, thus avoiding conflicts. However, all affected software needs to be rewritten to take this specification into account - a long and dire process with a nebulous future. In defense, many of the heavyweights such as GNOME, LibreOffice and the K Desktop Environment are fully behind the specification. Unfortunately, this does not help at all with older software and existing systems that do not follow the specification.

The solution presented in the remainder of this paper, the Host Profile File System, is built with the Filesystems in Userspace (FUSE) [1] framework and is considered superior to other possible solutions, both described above and not[7], because:

- It is transparent to the system.
- It does not rely on kernel changes.
- It does not require any changes to system services or programs.

[7]Like a redirfs [4]-based solution, with which a dotfile redirector could be implemented, yet would require additional kernel drivers and root access.

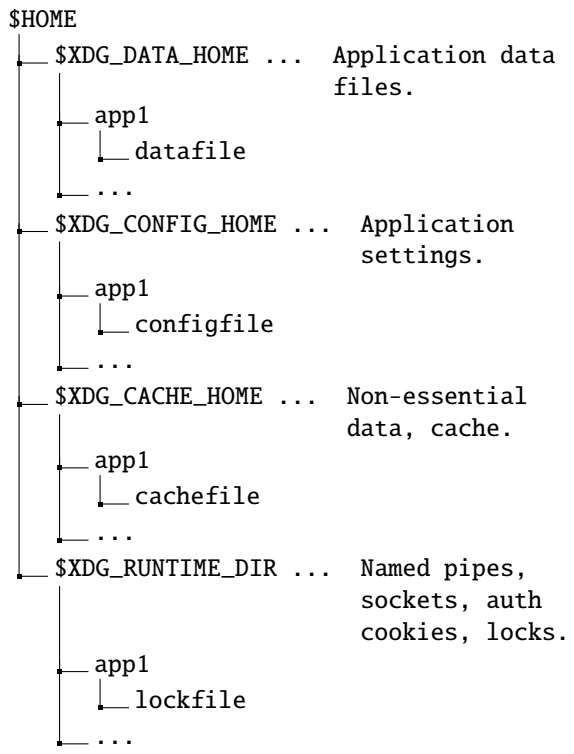

Figure 6: Simplified view of application data and settings under the XDG Base Directory Specification.

- It does not even require root access on the machine to enable.
- It is not based on fragile interfaces, or have machine dependence.
- The performance penalty is minimal.
- It is reasonably portable.

4 FUSE

FUSE is a Linux kernel driver that provides the necessary glue to have a fully user space implementation of a file system. This takes much of the complexity out of implementing a file system, due to not having to worry about complex locking and memory management interactions. FUSE also has a stable and OS-agnostic interface with consideration for backwards compatibility, meaning that maintainability is not an issue. FUSE has been ported to other Unix systems such as Solaris and OS X, which makes a FUSE-based solution viable in heterogeneous environments. Because a FUSE-based file system looks just like any other kind of VFS-provided file system, applications access it transparently. And most importantly, FUSE allows a user to mount their own private file system, meaning that neither system changes nor root access is necessary.

There is some overhead associated with the lack of direct `mmap()` semantics and with copying data between the FUSE driver and FUSE daemons. Tests with a pass-through FUSE file system have shown a 2% overhead [6]. A likely more realistic local test run, involving copying 22GiB of various software repositories, has shown a difference of under 9%[8] in total time spent, which is reasonable, given the end goal of accessing an NFS-mounted home directory, which wouldn't be used for I/O intensive workloads or large files anyway. FUSE overhead and performance has been critically analyzed elsewhere [9] with similar results.

5 The Host Profile File System

The Host Profile File System (HPFS) implements the previously described dotfile redirection design as a filter file system, mounted over the user's home directory. HPFS is implemented as a FUSE file system, and runs as a daemon with user privileges. The daemon forwards all file and directory accesses to the overlaid file system, and is able to do so by capturing the file descriptor of the user's home directory prior to the actual mount operation and by leveraging the `*at()` series of system calls to perform file I/O relative to an open file descriptor (See Figure 7). Without the `*at()` series of system calls this would not have been possible.

Incidentally, in a FUSE-based design we lose most of the complexity arising in a `LD_PRELOAD`-based solution, as all paths passed by FUSE are absolute (see Figure 8). Handling `readdir()` is slightly tricky, due to the need to hide existing dotfiles in the home directory root, and the need to account for dotfiles inside the redirection target directory. The current implementation does not virtualize the `structdirent` offset field, so applications relying on caching directory entries returned by the `readdir()` system call may see unexpected behavior. This limitation is not FUSE-specific, and would need to be addressed even if HPFS functionality were to be implemented as a VFS extension similar to GoboLinux GoboHide [10], which allows hiding files and directories from `readdir()`.

The redirection directory is passed to the HPFS daemon as a command line parameter, and is meant to

[8] 31m51s versus 34m56s.

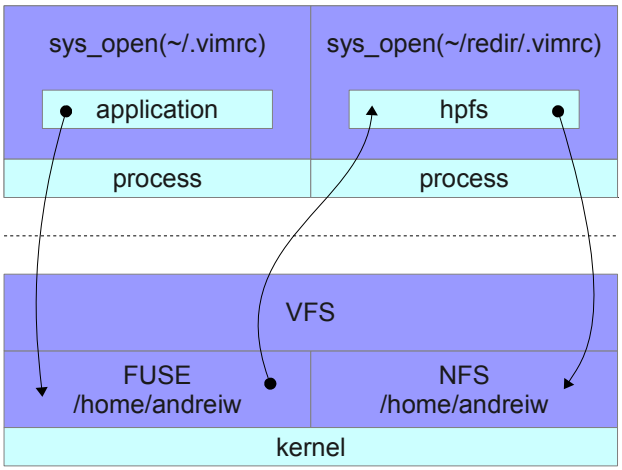

Figure 7: HPFS in action. HPFS is mounted over the already NFS-mounted /home/andreiw, hiding the original files from the user.

```
static int hp_open(const char *path,
  struct fuse_file_info *fi)
{
  /*
   * priv.fd contains the real $HOME
   * priv.redir_fd points to where
   * dotfiles are redirected to.
   */
  int fd = priv.fd;

  if (*path == '/')
    path++;

  if (!*path)
    path = ".";
  else if(*path == '.')
    fd = priv.redir_fd;

  fd = openat(fd, path, fi->flags);
  if (fd == -1)
    return -errno;

  fi->fh = fd;
  return 0;
}
```

Figure 8: `open()` handler for HPFS.

be derived by the support scripts. Two support scripts have been developed, one for the Bourne-Again Shell (`.bash_profile`), and one for the X11 Window System (`.xprofile`), to support redirection on both console and GUI logins. The scripts figure out the redirection path based on the host name, and enable the HPFS daemon if need be, while avoiding race conditions. The current version expects the system and CPU architecture to be the same everywhere, and a more complete version could thus be more intelligent in the choice of HPFS binary to run.

6 Conclusion

HPFS is fully functional and transparently usable in a real environment, and has been in active use for the past seven months across several machines. Further improvements would be improving `readdir()` virtualization, extended attributes support, filtering redirection by effective user ID (EUID), porting to other Unices and improving the surrounding ecosystem of scripts and helpers. Additionally, further performance impact measurements need to be done with HPFS and NFS. HPFS is open source, and the sources are freely available [12].

7 Acknowledgments

Many thanks to Alexandre Depoutovitch for his feedback on the paper, as well as Edward Goggin and Aju John for their support.

References

[1] Filesystem in userspace, 2011. http://fuse.sourceforge.net/.

[2] W. Bastian, R. Lortie, and L. Poettering. XDG Base Directory Specification. http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html.

[3] Fakeroot-NG. Ptrace ld_preload comparison, 2009. http://fakeroot-ng.lingnu.com/index.php/PTRACE_LD_PRELOAD_comparison.

[4] F. Hrbata. RedirFS. 2007. http://www.redirfs.org/docs/linuxalt_2007/paper.pdf.

[5] Jörg Zinke. System call tracing overhead, 2009. http://www.linux-kongress.org/2009/slides/system_call_tracing_overhead_joerg_zinke.pdf.

[6] S. A. Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh. stdchk: A Checkpoint Storage System for Desktop Grid Computing. 2008. http://arxiv.org/pdf/0706.3546.pdf.

[7] Linux Programmer's Manual. openat(2), 2009. http://man7.org/linux/man-pages/man2/openat.2.html.

[8] Microsoft Corporation. Managing roaming user data deployment guide. August 2006. http://technet2.microsoft.com/WindowsVista/en/library/fb3681b2-da39-4944-93ad-dd3b6e8ca4dc1033.mspx?mfr=true.

[9] A. Rajgarhia and A. Gehani. Performance and extension of user space file systems. 2010. http://www.csl.sri.com/users/gehani/papers/SAC-2010.FUSE.pdf.

[10] L. C. V. Real. Gobohide: surviving aside the legacy tree, 2006. http://www.gobolinux.org/?page=doc/articles/gobohide.

[11] M. Torrie. [uug] Gnome vs KDE, 2009. http://uug.byu.edu/pipermail/uug-list/2009-March/002134.html.

[12] A. Warkentin. Host Profile File System source repository, 2012. https://github.com/andreiw/HPFS.