Upperware: Pushing the Applications Back Into the System

Gorka Guardiola, Francisco J. Ballesteros, Enrique Soriano

Rey Juan Carlos University, Spain
{nemo,paurea,esoriano}@lsub.org

ABSTRACT

It is quite difficult and tedious to share devices among different operating systems. If we also want to share other resources, like the state of a web browser or an editor, it becomes next to impossible.

Similar problems are solved inside Plan 9 [14] and Inferno [4] by using the 9P protocol [9]. The normal approach, though, is to write an application or a device driver providing a filesystem interface. Our problem is somewhat different. We already have native applications like Word or Firefox. How can we use these applications, native to several operating systems, and at the same time have the ease of communications provided by 9P?.

In this paper we propose a simple way to do it: Wrap the applications and drivers with a controlling filesystem running on Inferno, hosted on the relevant machines. Then, export and share the filesystems, exporting them even to the local host system through some protocol it understands. Without much configuration, the user can print and read documents simply by using drag and drop at any of the involved machines. We propose the name *upperware* for this approach, which tries to abstract applications instead of the underlying system.

1. Introduction

The idea of Octopus [3] is to centralize the state of the applications in a computer that we call the PC. Then the user can run the terminal software which exports local resources as filesystems to applications running on the PC. Local resources may be applications and devices running at the terminal. In order to integrate these applications with the rest of the system we had to wrap them with filesystems, aggregating some attributes to them so they could be selected automatically. We have found that this approach is very simple yet powerful and lets users share resources easily without much configuration. We call it *upperware*. By proceeding further and exporting the resulting name space back to the underlying host OS, we reach a portable way to integrate the heterogeneous terminal, in a transparent way. By carefully thinking the filesystem interfaces, so that they can be used by copying files, we can convert this approach into a general solution even for non programmer users, which can do most things by drag-and-drop.

We have applied the *upperware* principle by writing *'device*drivers' for high level applications and services available on various systems we use. This includes printing, document viewers, voice synthesis, user activity monitoring, and a web browser service (still underway). In this article we describe such 'device drivers', their interface and what can be gained by using upperware: seamless communication and easy of use across heterogeneous platforms. We feel that upperware can be useful in general to integrate different platforms, taking the place of typical object-oriented middleware (OOM) approach.

Note that unlike in middleware systems, we wrap the namespace and make it available to the underlying system (at each terminal) in a transparent way. For the host the name space is just another file volume. However, it provides all sort of services for the user. Being natively available, native applications are able to use such name space as the user sees fit.

2. Organization

Experience with Plan 9, Inferno and Plan B [2] taught us that exporting devices and applications as synthetic filesystems makes it easy to integrate them into a distributed operating system. Applications like `acme(1)` [8] and `rio(1)` benefit greatly from programability by exposing a synthetic filesystem.

But our problem and approach is not quite the same, even if we still want to expose software resources from the underlying operating system and integrate them into a name space. Inferno does this to some extent with its devices while running hosted. The main difference is that Inferno places itself *side by side* to the host operating system, that is, it provides its own distinct virtual platform. Instead, we try to place ourselves above all the software running natively, meaning that we try to take advantage of the underlying operating system as much as possible, including also some of the applications.

What we are trying to accomplish is similar in approach to what Inferno does with the underlying filesystem of the host and to what 9vx does with the TCP/IP stack. We are trying to wrap software resources with a filesystem but using all the mechanisms provided by the host to ease programability and to interact with the user when necessary.

When trying to wrap underlying applications, we have to be careful to distinguish between two types of interfaces (meaning the semantics we assign to the filesystem): *passive* and *active*:

1.Passive interfaces wrap devices that are similar to low level devices. They do not require interaction with the user via the underlying system. In a sense they are data sinks and sources. Examples of this are the Octopus voice synthesis and printer devices. The fact that they do not require interaction with the user is very important, because it places restrictions on the way they should behave. For example, if the user imports a printer from a nearby computer and sends something to print, she does not want the remote host OS to ask her what paper size to use in the remote computer. Instead, it is more convenient if the file system interfaces provides an agreed-upon standard configuration and some means to change it.

2.Active interfaces wrap applications which do require interaction with the user. They are normally conduits of data or, at least, control the data flow. Examples of this kind of interface are editors and web browsers. An editor takes some file, changes it and puts it somewhere else as dictated by the interaction with the user. This interaction happens in the underlaying host system of the computer providing the service.

Orthogonal to this classification, but also important, is that for some of these devices it may be desirable to keep their state across computers and/or sessions, which affects the implementations of the servers to be provided (e.g., editors would copy the files, to remain autonomous).

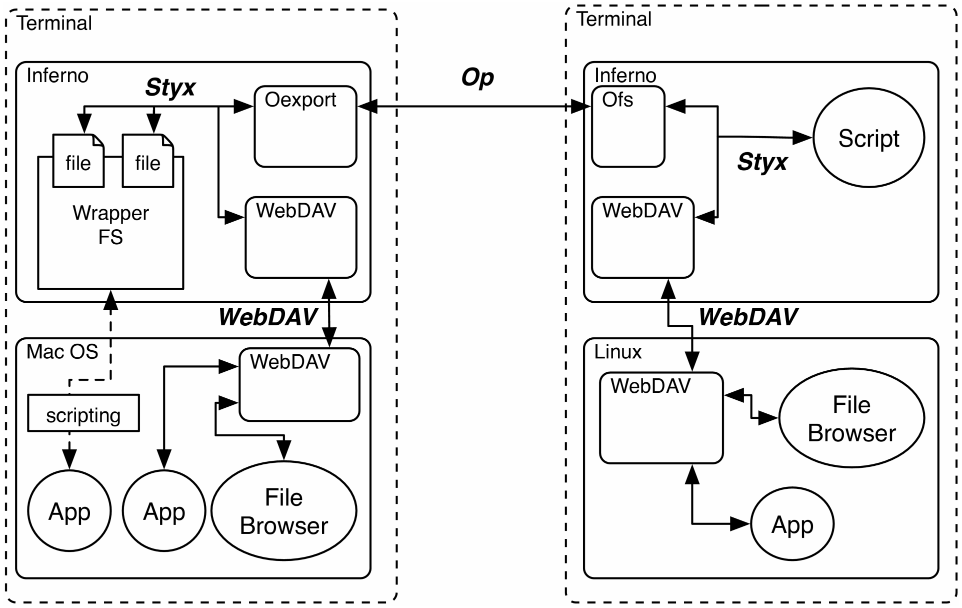

Figure 1: *Upperware organization.*

The filesystems we share in this fashion need some attributes to let the user select them appropriately by their location, operating system, etc. By convention, we add a file named `ndb` to the root of each filesystem. This file contains a list of attribute/value pairs describing the system exported. Such information is also kept at a central registry, for the user to look. In this way, the filesystem carries within itself its own description, should the user need to know.

3. Spooling

One thing we have found while trying to devise ways to expose and interact with the synthetic filesystems, is that *spooling* is a powerful strategy. By *spooling* we do not refer just to the classical spooling interface, because for us the directory doing the spooling is synthetic and some magic may occur in it. Nevertheless, the main idea remains the same: Files are copied to a directory and things happen to them when their turn comes. The resulting mechanism is similar to Apple droplets [1], as far as the user is concerned. Copying a file to a directory triggers an action.

Spooling also provides a way to interact with the filesystem through the host OS. Spooling directories are exported back to the host system using, for example, WebDAV. A simple drag and drop may be used to print a file or to edit it, hiding to the end user the details of the hosted system providing the mechanism.

Besides, spooling provides a way to structure the software. A whole filesystem does not need to be implemented for each different service (device or application); just some functions implementing the action performed to the files inside the spooler.

In the current implementation there is a portable module implementing the spooling filesystem. For each different *spooler*, an architecture dependant implementation of a spooler module provides an interface to the actual service. This interface has three operations, `start` `stop` and `status`. When a file has been copied (detected by the *clunk* on a fid open for writing) `start` is called. In some circumstances, reading a file also triggers a call to `start`. This makes the operation to be performed automatically when a file is copied to the directory. The `stop` operation gets called when a file is removed from the directory. It provides a mean for the user to abort or cancel a spooling on progress.

Using this simple scheme, we have a portable interface which can be used right from the file browsers provided by most (all?) host operating systems as well as a simple way to implement new servers and a straightforward user interface, well known by any user no matter the system employed.

Two example of spooling servers, discussed next, are the *view* and *print* devices.

3.1. View

View is an implementation of the spooler interface, i.e., provides a spooling device for viewing files and documents. It relies on the generic *open* command of the host OS to open the relevant file when its `start` function is called. For example, the device uses `gnome-open` in Linux, `open` in MacOS, and the plumber on Plan 9.

The device is intended for reading documents, like PDF or similar formats, in a passive way, although it is a little more general than that. For example, copying an MP3 file would reproduce it on the default player and, in a similar way, can be used to display images, play video files, open documents and web pages, etc. But note that edition of the browsed files is not supported by the device.

3.2. Print

Print is the interface to the printer system of the host OS. It is also a spooler module. A filed copied onto it is printed like in an old fashion printing spooler system. At the moment it uses the (CUPS) lpr commands in Mac OS and Linux, and lp in Plan 9. It prints just to the default printer with the default options, without any possibility of configuration. There has also been an ad-hoc printer module for Windows.

Even though this module is quite naïve, it has proved to be highly useful. Using it, it becomes trivial to do things like print to the closest printer (selected automatically via the ndb) from different locations and different operating systems.

4. Voice

This is a simple device, not based on the spooler. *Voice* exports mostly a file (apart from the `ndb`) which can be written to. Any text written into it is synthesized as voice. This is used mostly for messages of the system, but it can also be used for messages among users, like on Plan B.

Used with care (of not annoying the user by frequent messages), it is a useful complement for other interfaces to the system. For example, long termed commands, meaning those that do not complete after several seconds, generate a complete voice message to reassure the user that the command has finished.

In Mac OS we use a dynamically generated AppleScript script. For Linux we use *esspeak*. On Plan 9 we rely on the Plan B device for voice synthesis, which is actually relaying the work to any near-by Linux machine equipped with Festival.

5. Idle

In order to do things automatically, our system needs to know if the user is idle, meaning if he is using the terminal or not. Of course, we also need to know *which* terminal has been used last, to locate the user. This is done by employing some heuristics on data collected from several interactive applications and from the system I/O statistics. The *Idle* device is responsible for collecting such data and updating agreed-upon files describing the activity of the user on all her terminals. The rest of the system relies only on the portable files (and events) and does not need to be concerned about platform-specific details.

6. Web

We are currently developing a device for the web browser, *browserfs.* The first prototype offers three synthetic files to pull data from the web browser (`open`, `history`, and `bookmarks`) and a `ctl` file to push data and perform basic operations on the browser. All pull files are read-only and exclusive-open.

The three pull files provide their data in a canonical format. Different browsers use different formats to store data (mainly XML), but this is hidden by the provided interface. Browserfs offers the information using plain text files, in order to make it easy for humans to read it and for programs to transform it.

When a pull file is opened, the device retrieves the corresponding data from the host's browser by executing certain native programs on the host.

The *open* file offers the list of URLs that are currently opened in browser's windows and tabs, one per line. It corresponds to the pages being viewed presently.

History provides the last 100 entries in the browser's history record. Each line is formed by the date of the entry (seconds since epoch), the URL for the entry, and the title of the HTML page, separated by blanks. This format is easily tokenizable, because the date and the URL cannot include blanks.

The *bookmarks* file provides the list of bookmarked pages, as contained in a particular folder in the browser interface. The user is in charge of creating any new bookmarks in this folder to make them available from all his octopus terminals. This way, the user is able to select which bookmarks are shared among terminals and which ones are not. As it could be expected, the format of the *bookmarks* file is simply one bookmark per line, formed by the URL and its description.

The *ctl* file is used to push data and to perform control operations on the terminal's browser. So far, it implements only two commands: *open* and *close.* The *open* command executes the browser if it is not yet executing, makes a new window, and opens the given URLs in tabs. For example:

```
echo open http://google.com http://lsub.org > /term/browser/ctl
```

The *close* command forces the browser to close its windows and quit.

Together with *browserfs* we provide several scripts to capture and recreate the state of the browser.

The script *bookmarks.sh* reads the bookmarks from the user's terminals (i.e. */pc/terms/*/browser/bookmarks*) and merges them on a single file, that is automatically opened by the browser of the terminal in which it is executed. The same is done by *history.sh* for history entries, offering a HTML file with entries ordered by date.

Although this simple approach works for providing the web browsing state to the user, it is not enough to create a full illusion of using the same browser at all the terminals. Several extra control operations to update the browser's idea of bookmarks and history would be needed (not to talk about cookies).

In any case, the current version as described provides a portable implementation.

7. Mobile devices

For some devices it may be desirable for their state to follow the user. For example, we might want to keep the set of web pages being viewed the same, no matter the terminal. For other devices or terminals we may not want this to happen. Only the user knows the appropiate thing to do.

We try to provide this facility in a simple way, leaving it up to the user the choice of when and for which services it is to be employed. The overall scheme is described next, but we it is to be noted that this facility is still in the early stages and is not yet available.

We will try to keep this description concrete to the example of the browser, though the ideas are easily generalizable. We are currently working on this implementation of the browser. Before the user turns off a terminal, a shell script, *dump.sh,* is executed. This script stores the data of the terminal's browser in real files, in a well-known directory of the PC. When the users turn on a terminal, another script, *restore.sh,* is executed. This script reads the files in the well-known directory and recreates the state in the terminal's web browser. Another script, *followme.sh,* can automatically dump and restore data when the user location changes (i.e. he moves from one office to another). We will experiment with *browserfs* in order to create common policies for other application wrappers.

8. Implementation

The implementation of the machine dependant modules is a mixture of shell scripts, applescripts, C and whatever the host system might provide to do the job at hand.

Some of the operations that have to be done in the underlying system are trivial to implement, while other are annoying. It all depends on the interface offered by the application considered and the set of tools available.

For example, Applescript on Mac OS X provides a reasonable interface to control most applications. For instance, the Safari API provides operations to deal with tabs, URLs, and execute Java Script over a document, so some operations are easy to implement. On the other hand, Safari does not contemplate bookmark and history manipulation by third parties, so we have to deal with Mac OS property list (XML) files by hand.

We try to keep the number of features to a minimum so that the implementation is easy to write for all the operating systems involved as hosts for the Octopus. In the cases in which we can avoid most of the interaction with the native applications by relying on files instead, we do so.

This way, it is easier to keep up with the idiosyncrasies of the different systems, versions and applications involved. We also try to use the default or most popular applications of the system if there is more than one. For example, in Linux we are sticking whenever possible to applications which come by default with Ubuntu and Gnome.

To give a taste of how simple it is to write upperware for applications in Mac OS we will give the SLOC for the relevant parts in the Octopus. Most of the code is Limbo, with some scripts and applescripts generated at run time for the MacOS dependant part.

Tables 1 and 2 show the lines of code needed to implement the wrappers themselves, including the spooler (portable and Mac OS modules, respectively). Table 3 shows the lines of code needed to implement the Mac OS scripts of the current version of *browserfs.* Table 4 shows the lines needed to implement the infrastructure to be able to reexport the fileservers to the local host and to select the appropiate tree automagically.

lines	**module**
288	*browserfs.b*
277	*spool.b*
110	*view.b*

Table 1: *Lines of code, portable modules.*

lines	**module**
94	*idle.b*
159	*mbrowser.b*
82	*print.b*

Table 2: *Lines of code, Mac OS modules.*

lines	**program**
269	*browserfs.scpt (applescript)*

Table 3: *Lines of code, host commands.*

lines	**program**
1662	*webdav*
177	*watcher*
751	*mux*

Table 4: *Lines of code, infrastructure to reexport the fileservers to the local host.*

9. Related work

Middleware commonly sits between the application and the operating system and tries to abstract the operating system. We are trying to do the opposite, and abstract the application to make it available to the system (i.e., to other systems).

We are by no means the first ones to wrap local devices of the host OS and to export them using a lower level API. Npfs [6] and 9vx [5] wrap the local TCP/IP stack, for example with a filesystem. Inferno, `drawterm`, and many virtual machines like VMWare [12] [13] are able to export the local filesystem and some devices by using the host os. This is a common technique used in paravirtualization. The main difference is that we are trying to wrap high level devices and applications.

Filesystem like clients like `ftpfs(4)` or `sshfs(4)` try to wrap servers using filesystems as clients. The idea is similar to what we are trying to achieve, but they are remote servers. Here the application would be the client and it is built into the filesystem. We are wrapping local applications instead, also with a filesystem, in order to export them. In a sense, it is the reverse strategy.

Some early rudimentary attempts have been made before to export applications like a browser using plumber and ssh see [7]. As far as we know no one has tried wrapping them with a full fledged filesystem, which is much more rich in its interface and possibilities.

Regarding web browsing, there are utilities to manage and share bookmarks, such as Del.icio.us [10]. There are also programs to merge and translate bookmark files from different browsers [11]. None of these systems provide mechanisms to automatically recreate the state of the browser in different machines which is finally our goal. Also, they are meant just for a single particular service (web browsing) and not for all other tools needed by a computer user.

10. Conclusions

Our experience with the Octopus has made us realize how easy can it be to use applications once they are wrapped up and integrated back into the system interface. Something as simple as the print device which accounts for less than 500 lines of code including the spooler filesystem enables the user to print from Linux to Mac OS and viceversa (something that, perhaps surprisingly, we could not do due to incompatibilities between different native systems involved). It is amazing that similar issues are still problems in practice due to different configurations, security, and version issues. With upperware this can be accomplished smoothly keeping the user oblivious to the magic glue hiding behind the scene. The same goes for most other devices and services.

11. Future work.

More applications are to be wrapped to have a complete set. The web browser prototype has been written for Safari, but there is not yet supported for the other main-stream operating systems. An editor and a music player filesystem need to be written as well.

As of today, the spooler does not listen to messages from the application to detect when the user is done with a file (e.g., when a view window has been closed). Some infrastructure is already available for this, but it is not being used. The problem is that this would suppose a more intimate relationship with the native application, making it harder to write/port the machine dependent part of the service.

Support for Windows is also needed, but none of the Octopus developers use Windows daily and, although basic services are available thanks to the portability of Inferno, much remains to be done.

References

1. Apple, Shiny Droplets, http://www.apple.com/downloads/macosx/productivity_tools/shinydroplets.html, 2008.
2. F. J. Ballesteros, G. Guardiola, K. L. Algara, E. Soriano, P. H. Quirós, E. M. Castro, A. Leonardo and S. Arévalo, Plan B: Boxes for network resources, *Journal of the Brasilian Computer Society. Special issue on Adaptable Computing Systems. To appear. Also in http://lsub.org/ls/export/box.html*, 2004.
3. F. J. Ballesteros, P. Heras, E. Soriano and S. Lalis, The Octopus: Towards building distributed smart spaces by centralizing everything., *UCAMI*, 2007.
4. S. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey and P. Winterbottom, The Inferno Operating System, *Bell Labs Technical Journal 2*, 1 (1997), .
5. B. Ford and R. Cox, Vx32: Lightweight, User-level Sandboxing on the x86, *USENIX*, 2008.
6. Npfs, Npfs project, http://sourceforge.net/projects/npfs, 2007.
7. R. pike, Message by rob pike in 9fans: My web browsing technique, http://9fans.net/archive/2002/11/529, 2003.
8. R. Pike, Acme: A User Interface for Programmers, *Proceedings for the Winter USENIX Conference*, 1994, 223-234. San Francisco, CA..
9. D. Presotto and P. Winterbottom, The Organization of Networks in Plan 9, *Plan 9 User's Manual 2*.
10. J. Schachter, Del.icio.us, http://del.icio.us, 2003.
11. E. Software, Bookit, http://everydaysoftware.net/bookit, 2003.
12. J. Sugerman, G. Venkitachalam and B. H. Lim, Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *USENIX Annual Technical Conference*, 2001, 1-14.
13. VMWare, VMWare, *http://www.vmware.com*, 2001.
14. Plan B User's Manual. Second edition., *Laboratorio de Systemas, URJC. GSYC-Tech. Rep.-2004-04.*, 2004.