

UpperWare: Bringing Resources Back to the System[†]

Francisco J. Ballesteros, Gorka Guardiola, and Enrique Soriano
Laboratorio de Sistemas
GSyC, Universidad Rey Juan Carlos
Madrid, Spain
{*nemo, paurea, esoriano*}@lsub.org

Abstract—If we expect the computer to vanish in the background, to make pervasive computing a reality, first we must be able to provide the illusion that all the user’s computers, devices, and applications are part of a greater, virtual, computer.

In order to build such a virtual computer, resources must be provided with a portable programmatic interface to interact with programs running on top of current mainstream operating systems. These interfaces should be understandable by the final user, in order to allow for their manual operation. Also, a system layer is needed to tie resources together, control them, and handle changes of context to permit adaptation.

In this paper, we present an architecture that permits this: the UpperWare. This architecture permits to abstract and export the computing resources offering an universal interface by using a synthetic file system scheme.

Keywords-middleware; operating systems; upperware; ubiquitous computing; pervasive computing; smart spaces;

I. INTRODUCTION

Surprisingly, in 2010, with all our computers connected to the Internet, it is difficult to bring all our resources together to get some synergy and to provide a personal pervasive environment.

Resources considered include (i) *devices* (e.g. printers, cameras, or speakers), (ii) *data* (e.g. data files or system properties), and (iii) *applications* (e.g. desktop applications, viewers, or browsers).

Using standard system support, if we try to use these scattered resources *together*, we encounter a plethora of pitfalls related to control and configuration. We have to be in charge of controlling all the processes for the task at hand and suffer the lack of system support for context information and adaptation. If our computers run different operating systems (and that is the common case, because of the diversity of mobile and desktop computers), the problem becomes worse. In the normal case we are not able to directly combine resources to perform the desired task. This applies to experienced users. For end users, difficulties include even sharing some resources that are expected to be shareable nowadays, such as data files. In general, scenarios described in pervasive computing literature are still *sci-fi* for end users. And we want this to change.

Regarding devices, some of them require some kind of lightweight administration to be shared between different computers. Even being lightweight, this configuration is a burden and adds complexity. For example, when we buy a printer, we usually have to manually add it to each one of the computers we use.

Even worse, other devices are built under the premise that they will not be shared, even when the usefulness of sharing them is obvious.

Regarding data, it is dispersed among our computers. If we want to make all our files available to all our computers, we have to configure a network file system server on each one. Besides the configuration burden, we also have to keep all the involved namespaces in mind while using them. Control version software, replication tools, and systems like Omnistore [9] relieve the problem. But again, control has to be in the mind of the user. This is not a pervasive computing environment and it will never be as long as this kind of problem persists.

Regarding applications, they can be also exploited in order to get valuable information and tools for pervasive environments. The local machine frontier may vanish if we keep the application state consistent among different computers. If we start our office desktop computer in the morning and it recovers the state of the applications as it was left in a home computer the previous night (e.g., a text processor editing a document and a web browser with certain bookmarks and history), then we start to perceive both computers as part of a greater environment and not as separate machines. The preservation of the state is the main reason for the popularity of web based applications, such as Google Docs. We believe that it is not necessary to run the applications in a machine provided by a third party just to keep the state of our *virtual computer*. We may keep the state in our own machines instead.

In this paper we describe the architecture behind a system, the Octopus, that addresses these issues. The system has been in use for more than two years and is being used to write this paper (both at some of our homes and at our offices, yet we feel we are using a single “pervasive computer”). The architecture is based on two ideas:

- 1) Control is centralized on a dedicated machine. We call this central node *The PC*. Other computers of the

[†] This work is supported in part by grant TIN-2007-67353-C02-02.

user are the *terminals*. The PC can be considered as a dedicated machine per user, a virtual machine offered by a cloud computing provider, or a black box, just like a router or a standalone network disk. In fact, any conventional computer can act as the PC if it runs Plan 9, Windows, Mac OS X, Linux or FreeBSD.

- 2) All resources (including not just devices, but also data and applications) are exported through file system interfaces that are available for any of the machines of interest for the user. This is called *UpperWare*.

UpperWare is software that makes devices, data, and applications available at the system level and remotely accessible from any system. Devices are exported as file systems, data is kept on file systems, applications and other miscellaneous resources are wrapped by tiny synthetic file systems following the approach of Plan B [1], [2], [3].

UpperWare abstracts high level resources instead of providing access to any underlying mechanisms. For example, considering a text processor as a resource, *UpperWare* offers an interface to open a document at a specified page. It does not provide an interface to edit a table contained in the document. Only features known to be available everywhere are made available through *UpperWare* interfaces. This helps providing a homogeneous system out of highly heterogeneous devices and computers.

Using *UpperWare*, it is easy to build an environment out of different systems. Every machine with resources of interest runs some *UpperWare* and exports its resources to the PC as if they were files. The PC provides a per-user global namespace that is shared by all machines of interest for the user. The namespace aggregates all resources and keeps them organized as hierarchy of files. Of course they are not files: they may be printers, applications, tools, data, etc. There are several components which manipulate this namespace to adapt the system to changes. For example there is a multiplexor (\circ/mux) which is used to select the appropriate devices for specific contexts. Because resources are available as files through popular protocols (e.g., WebDAV), all systems and devices are ready to use such resources, even without running software for that purpose (because all machines know how to share files in one way or another).

It is also trivial to write programs that rely on *UpperWare*. They do not need to use any special software at all! That is, besides the native interface used to read and write files. Once again, all programming languages know how to do that and no specific software is required. By carefully choosing names for resources, it is clear where to find the resources and what they are.

II. UPPERWARE

We call “*UpperWare* driver” the piece of *UpperWare* providing a particular resource (because we treat all resources as devices, even when they are applications).

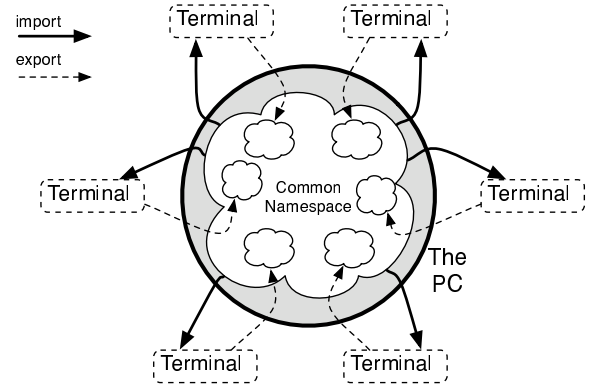


Figure 1. The terminals export their resources and import the common namespace.

In the system built to gain experience with *UpperWare*, the Octopus, a central registry is kept on a per-user central machine, the PC (each user has a dedicated PC). When a machine starts (or the user starts the Octopus on it) it connects to the user’s PC, registers all its resources exported through *UpperWare*, and also imports all the resources that are aggregated by the PC.

In our current implementation, an *UpperWare* driver is an Inferno synthetic file system implemented in Limbo [5] that wraps a resource of the underlying system. As explained before, an *UpperWare* driver offers a high level interface to access the resource driven.

There are two types of *UpperWare* resource drivers as we describe next.

- **Passive *UpperWare* resource drivers** process or generate data without any user interaction. For example, the voice driver does not need to interact with the user. It receives a text string and interprets it using a voice synthesis application to deliver the speech message. This resource is a data sink. There are other resources of this kind. For example, the printing service uses the host’s default configuration to print a document. Other passive resources are not sinks, but data sources. One example is the camera. This resource generates data when prompted; it takes the photo in a default format and provides the data. Passive resources can be kept simple by using default options, avoiding any kind of configuration or interaction.

A data sink resource can be wrapped as a *spooler* (a directory to place a file to be processed) or as a single write-only synthetic file, depending on the kind of the input data and on the latency of its operation. Data sources are commonly wrapped by a single read-only synthetic file because of concurrency control considerations.

A result is that most passive *UpperWare* resource drivers share much of their implementation. The main

difference between one driver and another is the underlying command used to implement the service (e.g., to print, to open a document viewer, etc.)

- **Active *UpperWare* resource drivers** are those requiring some form of user interaction (the user can be either a human or a program). They receive commands and respond to them, establishing some type of dialog. Interfaces for active resources are more complex than those for passive resources.

In this case, interfaces are tailored for each particular resource, which means that their implementations do not have much in common. This is the common case for drivers abstracting applications. Each application may be abstracted in a different way. In the worst case, we can consider the application both a data source and a data sink and let the user copy the input data in, operate on the application, and copy the resulting data out. However, more research is required here.

Independently of the type of driver, there are multiple mechanisms available to drive a local resource within an *UpperWare* resource driver:

- **Application execution** In some cases, it suffices to execute an existent application to provide the required service. For example, on Mac OS X hosts, the camera resource is provided by executing a simple command that takes a picture and generates a JPEG file as output. Another example is the voice device on Linux hosts, that is provided by executing a voice synthesis command named `eSpeak`.
- **Scripting** Sometimes a simple command is not enough, but a set of commands can perform the operation. A simple shell script on a Unix system can provide the desired action. On Mac OS X hosts, AppleScript provides considerable control over native applications. In this case, *UpperWare* is abstracting what AppleScript can do and providing a simple interface that makes other systems completely unaware of AppleScript. For example, voice synthesis uses a shell command on Linux terminals, but uses AppleScript on Mac OS X terminals instead. System components may ignore all this and just write to `/mnt/voice` or to `/mnt/term/terminalname/voice` whatever they want to speak.
- **Custom programs** In other cases, it may be necessary to implement custom programs to obtain the desired service out of the underlying host system. A native program can use the host system interface (libraries, system calls, etc.) in order to get the required data or perform the required task. *UpperWare* would then wrap and re-export the service as done on the first case.
- **Configuration files and databases** In many cases we can directly get the desired data out of the host file system. For example, an *UpperWare* driver can read

some relevant information directly from a specific XML configuration file and transform it into a conventional format in order to provide it through the *UpperWare* file interface.

- **UI manipulation** Some tasks can be performed by manipulating the GUI widgets of a native application. Scripting languages such as AppleScript are capable of manipulating an application navigating through their GUI elements and performing operations over them. This approach turns out to be too slow for repetitive operations and may disturb the user. For this reason, we have avoided doing so in our prototypes. However, it could be used to perform short, fast operations.

Each *UpperWare* resource may use one or more of these mechanisms to do the job. We describe next some *UpperWare* resource drivers that we have been using for more than two years.

III. UPPERWARE RESOURCE DRIVERS

In what follows we describe some of the *UpperWare* services. Each one of those described here is an example of other similar services that may be built along the same lines.

A. *Speech messages: /mnt/voice*

The voice device is the simplest device we have. It is used to deliver speech messages to the user. For example, when the user executes a remote command in the PC, the shell notifies the user when the command has finished its execution, and warns him about its exit status. Also, the mail application may deliver voice messages. In the same way, a context application describing users connected to the system may notify that certain users have become online/offline at particular locations.

Any program using the voice device delivers its message to a machine near the user, but may remain unaware of the actual device used. Context information about the user location is used by `o/mux` to make an appropriate voice device available at `/mnt/voice`. The same happens to services described on the following sections.

The voice device is a data sink, and its interface consists of a single write-only file, `speak`. When this file is written, the text string is processed with a voice synthesis program in the host system to reproduce the message through the terminal speakers. For example, a calendar application may deliver an alarm to the user by executing the following command¹:

```
; echo 'appointment at 5 P.M.' > /mnt/voice/speak
```

This code illustrates how simple this approach makes accessing the voice device for a programmer:

```
void  
deliver(char *msg)
```

¹All shell examples are expressed in the RC Shell language. *UpperWare* drivers are independent of the Shell.

```

{
    /* error checks omitted */
    fd = open("/mnt/voice/speak", OWRITE);
    fprintf(fd, "%s", msg);
    close(fd);
}

```

The Linux driver uses a command called `eSpeak` to process the text and deliver the speech message. The Mac OS X driver uses a different mechanism. It executes an AppleScript that uses the native interface for speech messages. As we stated before, both other software and humans may forget about this because *UpperWare* hides the implementation behind a file interface.

B. Image input: `/mnt/camera`

Input devices can also be wrapped and served as high level resources. The camera device is an example.

This device is a passive data source that exports a camera as a unique file. When the file is open, a picture is taken and the resulting data is supplied to reads of the file that follow the open request, with the picture encoded in JPEG format. When the file is closed, the picture is discarded.

For example, to take a picture from the camera at home and display it where we are we could execute this:

```
; cp /mnt/terms/hometerm/camera/data /mnt/view
```

C. Web browsing: `/mnt/browser`

The browser device is the first active device that we made. It is intended to control the web browser of a terminal, tolerating user interaction with the browser. The aim of this device is to control the web browser enough to replicate its state at other terminals. It provides a generic interface to control the execution of the web browser and perform simple actions to opened web pages, history entries, and bookmarks. This device must be understood as an example of how to design an active device, other applications may be controlled in a similar way through *UpperWare*.

The interface of this driver is more complex than the previous ones. It provides a directory with four read-write files: `ctl`, `open`, `bookmark`, and `history`.

When read, the `open` file provides the list of URLs that are currently open in the browser (both windows and tabs), one per line. The same kind of data (multiple URLs, one per line) can be written into this file to ask the browser to open URLs in a new window using tabs. For example, to get the list of open pages in the browser and open a new one, one could do this:

```
; cat /mnt/browser/open
http://mail.google.com/mail/#inbox/1208
http://www.lsub.org/who/index.html
; echo 'http://lsub.org/who' > /mnt/browser/open
```

The same happens to bookmarks and history management. For example, to create a completely new history combining all the histories from all the terminals we can read the history files from all the browser devices (from all the

terminals), sort them by the timestamp field, and then copy the result to the terminal we are using now:

```
; cat /mnt/terms/*/browser/history | sort -n > \
    myhistory.txt
; cp myhistory.txt /mnt/browser/history
;
```

In order to avoid race conditions, `open`, `bookmarks`, and `history` are exclusive-open files. That is, only one process can open them at the same time.

IV. A PERVASIVE COMPUTING SCENARIOS

Suppose that we want to post an alert if we detect any movement at home. We could use the following script:

```
# $hometerm is the home terminal's mount point
cat $hometerm/x10/livingroom
cp $hometerm/camera/data /mnt/view/photo.jpg
echo 'movement at home >> /mnt/voice/speak
```

The first `cat` would block reading a file that is the interface for an X10 movement sensor at the living room. Upon movement, that file may appear to contain the string `yes`.

The `cp` command in the second line achieves many things on its own. When the file `$hometerm/camera/data` is opened for reading, the camera of the terminal at home takes a picture. Then, `cp` reads the data of the picture and copies it into the file `/mnt/view/photo.jpg`. But this file is not a real file. The `/mnt/view/` directory is the view device of the terminal where the user is working. Thus, the picture will be displayed at the current user's terminal (on a Mac OS X terminal it will be opened by the `Preview` application). It all is accomplished by using an ancient command that only copies bytes from one file to another, which was never meant to do this kind of job.

The third line instructs the voice device to speak a warning message to alert the user. The voice device on `/mnt/voice/` may be actually located at another terminal near the user, as long as it shares its location with the user.

If the user detects an intruder at home, he could execute this other script:

```
# $hometerm is the home terminal's mount point
cp $hometerm/camera/data /mnt/print/intruder.jpg
echo vol 100 > $hometerm/play/ctl
cp /sounds/alarm.mp3 $hometerm/play/
```

The first line prints a new photo from the camera at home using the printer nearest to the user. The second line sets the volume of the player to the maximum value in order to scare the intruder away. The last line commands a music player driver² running at home to play an audio file that is an alarm sound.

As seen, the *UpperWare* offers powerful mechanisms to implement applications for pervasive computing. Files make resources readily available. `o/mux` selects resources used

²This driver is not described in this paper due to space limitation.

by default. The central name space coordinates and collects resources from other machines.

V. END USERS

All the mechanisms described in previous sections can be directly used through a file browser interface. The central namespace is re-exported to the host system using WebDAV (but there are other protocols available). Therefore, all virtual files and directories are available through the standard file browsing tool employed by the user. In the Octopus, all the environment seems to be a network file volume.

For example, to print the bookmarks of the browser at one machine in a printer attached to another machine, the user only has to drag the `bookmark` virtual file of the corresponding directory and drop it into the corresponding `printer` directory. Copying the file to the `bookmark` file of another terminal would replicate the state of one browser into another.

VI. IMPLEMENTATION

We have tried to keep the number of features of *UpperWare* resource drivers to a minimum so that is easy to write versions for different host systems. As a result, the implementation of current *UpperWare* prototypes is quite small. We used Limbo[5] as the programming language. Limbo is compiled to byte code and interpreted using a virtual machine called Dis, which is part of Inferno [5]. In this way the same implementation can be used for all popular operating systems.

An *UpperWare* driver usually includes two Limbo modules: a portable one and a system dependent one. A **portable module** is fully implemented in Limbo and usually implements a file server interface for the driver. This includes the implementation for `open`, `read`, `write`, and `close` operations on files provided by the driver. These operations may invoke functions of the **system dependent module** in order to access to do its job. This module may either execute a native program on a particular system, generate and execute a script, or rely on any other system interface that might be used to do the job, as explained in section II. The entire implementation of the Octopus is quite small. There are 16159 lines of portable code of core components of the system, such as the event delivery system and a name space multiplexer, and drivers. Non-portable modules span 2862 lines of code, including programs written in several languages for Mac OS X, Linux and Plan 9. The biggest driver implementation is the browser device for Safari, which is 878 lines of code (351 of Limbo and 494 of AppleScript).

VII. RELATED WORK

Plan B was a complete operating system and ran on bare hardware. Thus, all the terminals were forced to run Plan B instead of a mainstream operating system. On the other hand, *UpperWare* runs as application software. In the Octopus, it is

implemented using Inferno [5], which is available for most popular operating systems³.

Much of related work tries to accomplish some of the specific tasks used as examples in this paper. Several examples could be performed using a remote shell (e.g. SSH) together with a network file system. In fact, this is how advanced users try to face these situations in current computer environments. These users try to cross the frontier between their computers. **We want to break down the frontier.** The tasks described in this paper are just examples to illustrate the global goal of *UpperWare*: to export services as high level abstract devices to integrate them at the system level, and then bind all these devices together in a personal pervasive computer environment.

Some systems permit to execute code in the web browser. Google Native Code (NaCl) [11] is an example. It provides a sandbox to execute native X86 code in the client's web browser. *UpperWare* does not send any code in order to be executed in the client machines. In fact, NaCl is closer to the infrastructure *UpperWare* needs to execute (Inferno[5] in this case) than to *UpperWare* itself. Nevertheless, NaCl is not suitable to run *UpperWare*, because its sandbox is too restrictive.

Other research on system support for pervasive applications [7], [6], [8], [10] use traditional object-oriented middleware to export services. In this case, end users are not able to manually use the services, they have to use custom applications to do so. Moreover, custom applications must be implemented using specific libraries or frameworks. Instead, *UpperWare* integrates services at the **native OS system level** (the file system). Thus, custom and legacy applications, as well as end users, are able to directly use them.

In some pervasive computing systems, desktop applications are used as components for pervasive applications. For example, GAIA's [10] presentation application uses PowerPoint to open slides using its COM interface. iROS [8] also uses desktop applications as components for pervasive applications. Unlike them, *UpperWare* aims to generalize this kind of operation, hiding resource (devices, applications, and data) details behind a high level and abstract interface. Following with the presentation example, an *UpperWare* driver can use the COM interface as the mechanism to control PowerPoint in Windows hosts and shell scripting to control OpenOffice Impress in Linux hosts, while the presentation application remains unaware of these details. One important point is that the same applies to any application in the system, not just to the few new, pervasive, applications.

Some propose mobile agents that follow the user as a mechanism to support user mobility, see for example [12]. On the other hand, the *UpperWare* approach does not move any code but the state of replaceable components. If the

³Inferno is also an operating system that may run either on a bare machine or hosted on other popular systems

user moves, the system applies the corresponding state to the analogous components in the new location, and switches to use them, instead of using the ones at the old location or sending code around.

Other systems, such as MobiDesk [4], completely virtualize the user's session in a virtual machine provided by a cloud computing infrastructure. This way, the user can use his personal computers as mere terminals. These systems address the problem of keeping the system state among different sessions, but they do not address the pervasive scenarios described in previous sections. *UpperWare* does not virtualize processes and resources. It abstracts and exports them as replaceable components via an universal and well known mechanism (files).

REFERENCES

- [1] F. J. Ballesteros, E. Soriano, G. Guardiola, and K. Leal. *IEEE Pervasive Computing*, 3(6):58–65, 2005.
- [2] F. J. Ballesteros, E. Soriano, G. Guardiola, and K. Leal. *Pervasive and Mobile Computing Journal*, 2(4):472–488, 2006.
- [3] F. J. Ballesteros, E. Soriano, K. Leal, and G. Guardiola. Plan b: An operating system for ubiquitous computing environments. In *Proceedings of the IEEE International Conference on Pervasive Services*, pp. 126–135, 2006.
- [4] R. Baratto, S. Potter, G. Su, and J. Nieh. Mobidesk: mobile virtual desktop computing. In *Proceedings of the 10th annual international conference on Mobile computing and networking*, pp. 1–15, 2004.
- [5] S. M. Dorward. The inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.
- [6] S. D. Gribble, M. Welsh, R. Behren, E. A. Brewer, D. E. Culler, N. Borisov, S. E. Czerwinski, R. Gummadi, J. R. Hill, A. D. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Y. Zhao. The ninja architecture for robust internetscale systems and services. *Computer Networks. Special issue on Pervasive Computing*, 35(4), 2000.
- [7] R. Grimm and B. Bershad. Future directions: System support for pervasive applications. *Proceedings of FuDiCo*, 2002.
- [8] B. Johanson, A. Fox, and T. Winograd. The interactive project: Experience with ubiquitous computing rooms. *IEEE Pervasive Computing*, 2:71–78, 2002.
- [9] A. Karypidis and S. Lalis. Omnistore: Automating data management in a personal system comprising several portable devices. *Pervasive and Mobile Computing*, 3(5):512–536, 2007.
- [10] M. Roman, C. K. Hess, R. Cerqueira, A. Ranganat, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 1:74–82, 2002.
- [11] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.
- [12] P. Yu, J. Cao, W. Wen, and J. Lu. Mobile agent enabled application mobility for pervasive computing. *Lecture Notes in Computer Science, Ubiquitous Intelligence and Computing*, 4159:648–657, 2006.