The Octopus: Towards Building Distributed Smart Spaces by Centralizing Everything

Francisco J Ballesteros, Pedro de las Heras, Enrique Soriano, and Gorka Guardiola
Laboratorio de Sistemas. GSyC, Universidad Rey Juan Carlos. Spain.
Spyros Lalis
University of Thessaly. Greece.

Abstract

The world changes fast: today, most users typically have access to several computers at work, at home and while in transit, all of them interconnected through Internet. This world is far from perfection: users face decentralized, uncoordinated, heterogeneous, and highly dynamic, practically uncontrolled environments. It is hard to develop and deploy applications for such a world. One of the aims of pervasive computing is to be able to build applications that could be used from anywhere, and than could exploit resources available anywhere. That is, today, it is difficult to use different resources smoothly, unless one sits in front of the computer where they were deployed. We argue that one way to solve this problem is to first apply centralization as a design principle, and then recognize the existence of other computers running their native OS's, importing their devices and controlling them from a *central computer*. Doing so it would be possible both, to easily build systems that can leverage on all the devices available to a given user, independently of its location, and to easily use those systems. Our preliminary work, derived from the Plan B OS, strongly suggests so. This paper describes the evolution of Plan B, called Octopus, designed along this principle.

1. Introduction

One of the problems faced while developing applications for smart spaces and other pervasive computing environments is that the underlying computing platform (indeed, platforms!) is highly heterogeneous, dynamic, and complex.

Today computing environments are complex, if not chaotic. They are made of a myriad of devices and machines interconnected through multiple networking technologies. They are subject to network partitions, have administration problems, have different capabilities, run different sets of system software and applications, and the list goes on. In 2007 we suffer, more than ever, decentralized, uncoordinated, heterogeneous, and highly dynamic, practically uncontrolled environments. The net effect is that users have at hand more hardware and more powerful than ever, but can not exploit this computing potential in an easy way. They have to explicitly indicate what computer they want to use for each thing done, what finally causes a nightmare, because **all the control is in the mind of the user**, instead of being implemented by the system. The net links all the computers, but many times the services of a given computer can not be used if not sat in front of it. Something is really wrong in the systems software we use.

During the last decades many distributed computing solutions have been proposed by the research community. A non negligible part of them were solutions to problems that arise only when considering all computers as equals. The complexity introduced by this assumption leads many times to solutions that are intrinsically complex and bad performing. We believe that the fact that computing resources are scattered should not preclude centralized solutions.

Google [2] is a good example that can be used to support this not so radical claim. It centralizes the implementation (at least as seen from the outside) while providing ubiquitous access to it. Users open their browsers, address them to *pages.google.com*, *docs.google.com* or to *calendar.google.com* and start working. The Google file system [2] is another example. It centralizes control to handle with a much more simple and centralized implementation the myriad of distributed storage and computing devices used to implement the service. It builds a distributed file system by centralizing its core algorithms and control, and distributing data. The same underlying idea could be applied to most of systems software.

We claim that on a globally connected world, it is not cost effective to consider all computers as equals. At the end this introduces additional complexity in the system. This, we hypothesize, could explain at least in part why this kind of distributed systems solutions have not been adopted by the industry. In

order to escape from complex solutions, it is centralization the first design principle that must be applied to solve the many problems that users universally face. By doing so we expect to eliminate most uncoordination, uncontrollability and inconsistency from our systems.

Systems like WebOS [10] and Protium [11] tried to face these problems. Arguably, it seems they failed. But we argue that we, as the research community, just have to try harder on the path toward centralized solutions, if we do not want this kind of software to be invented outside our community, as it happened with the Web!

2. System Model

The *Octopus* is a system being built to provide a supporting platform to build distributed smart spaces and to provide pervasive applications, that could be reached from anywhere, and that could use resources from anywhere.

The system model used to build *the Octopus* has been designed assuming that in the next ten years all computers of interest will be connected to an IP network, being all networks interconnected, with links between networks potentially having bad latency, but usually having enough bandwidth (a minimum of several Mbps). With respect to computing capacity we assume that the power of a single computer suffices for most if not all the tasks of interest for the user, what makes the rest of computers available just a repository of additional devices for *the computer*. Finally, we assume that all the user wants is to use his/her devices with the computer, independently of his/her location, launch applications on the computer, and store data on it.

In *the Octopus*, there is a **single dedicated computer per user**, *the computer*. It does not have any input/output resources of its own; we think of it as a box with lots of (virtual) memory and processing power. I/O devices are considered to be attached to the computer via the network. All user programs execute on the computer, independently of the user's location and of the devices and resources required to run them. Google services have distributed implementations, in part, because they have to scale worldwide. In the Octopus, the computer is for a single user, which obviates the need to scale. In any case, devices and services may be shared among different users (eg., by attaching them to more than one *computer*).

The system not only encompasses *the computer*, but may span all the distributed devices of interest. These devices are usually attached to other computers. We do not care which operating system and/or applications run on machines that are not *the computer*. In our view, such software is not different from a hardware device that we plug into the computer. Only that such device may just be able to do elaborate things (e.g., decode and reproduce video). For the computer, **Internet is the system bus**, and other systems (and all their software!) are just more hardware for the Octopus. Even though this may seem inefficient, our previous experience suggests that it is both doable and desirable, as justified later.

While devices can be highly heterogeneous, are distributed, mobile, and can be switched on and off at any time, the computer is a single, central, homogeneous system where all the system software runs. This means that the interfaces between resources and the central part of the Octopus must be of a high-level of abstraction and must be designed taking into account one of the main assumptions we made: links can have bad latency. Our approach is to map these highly abstract interfaces into virtual file trees, following the Plan 9 [7] and Plan B [1] approaches.

Devices for the Octopus have a high level of abstraction (as they did in Plan B). For example, an audio device accepts MP3 files as input, not low-level PCM samples. The audio device exports a file tree with two files as its interface. The `output` file accepts MP3 encoded data for playing. The `volume` file accepts (and reports) text strings to control (and check out) the volume, balance, and other features of the device.

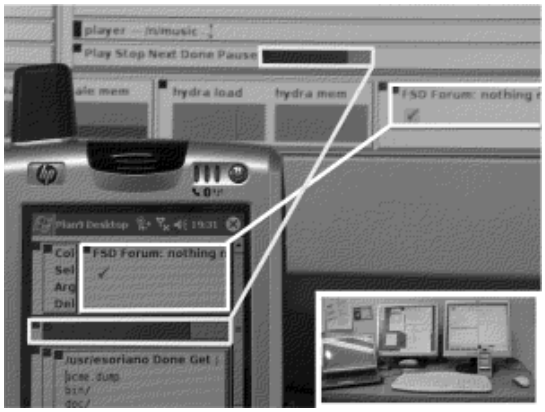

Figure 1: *Distributed UIs, centralized implementation.*

As another example, the window system provides as its interface a file tree. The tree of files represents a tree of widgets shown at a screen. All editing and user interaction happens within the window system and applications receive high-level events, not mouse and keyboard events. They operate on their interfaces through the files provided by the window system. As a

result, the needs regarding latency and bandwidth of the communication link between the window system and the application are much lower than for other systems.

3. Resources and name spaces

The Octopus running at the (central) computer is similar to a Plan 9 [7] or Plan B system [1]. It provides per-process name spaces to let the user customize the environment seen for different applications.

Devices and resources plugged into the computer (via the net) are registered using a centralized registry service. All of them correspond to (virtual) file servers that provide a file based (abstract) interface for the resource considered. Each resource is identified by a global name (eg., `/audio`) and a set of attribute/value pairs (eg., `loc=room136.urjc` to indicate the known location for the resource).

A name space mount request arranges for a particular name (e.g., `/n/audio`) to refer to a device meeting a certain criteria. This is similar to Plan B name spaces [1], where a command like

mount '/audio loc=home user=nemo' /n/audio

mounts any device registered with name `/audio`, located at `home`, and owned by `nemo` at the name `/n/audio`.

If no such resource exists, the system arranges for `/n/audio` to look like an empty directory. If a resource being used vanishes, open file descriptors report I/O errors, and the system tries to select any other registered resource that also meets the user requirements. While no such resource exists, the directory seems to be empty.

The name space mechanism is the same we used for Plan B, as described in [1]. However, in the Octopus, the system is no longer distributed and all the name spaces are kept in *the computer*. Each name space is the glue that keeps together the devices and services (ie., the virtual file trees they provide as their interfaces) used by the application. In order to be able to integrate (to some extent) machines running other operating systems, the file tree as seen by the central computer may be also exported to other machines using either 9P, NFS, or CIFS.

4. The Octopus prototype

We are implementing the Octopus by modifying our OS, Plan B, along the ideas presented here. To illustrate our approach, and to justify that it can work well, we describe how the Octopus works for a particular application, a music player (see figures 1 and 2). We focus on the implementation of the window system of the Octopus, and in the way we implement applications that use remote devices, using the audio device as an example. All the software involved and mentioned here has been already implemented and is functional.

To provide a user interface, the player uses `o/mero`, the window system of the Octopus. It can create widgets like volume-gauges, text panels, and controls by creating directories in the virtual file tree provided by `o/mero`. Following our model, both the player and `o/mero` run on *the computer*. But the audio device and the screen(s) could be far away.

Unlike in the previous incarnation of `o/mero`, implemented for Plan B [1], the Octopus window system does *not* draw anything. It simply provides the virtual file tree representing widget hierarchies, accepts file operations to operate on the widgets, and notifies the application of high-level UI events. The main events are *look* for something and *execute* something. What "something" means, depends on the application. The UI service handles that argument as a string. The application and the user know the meaning, the system does not.

User interfaces created in `o/mero` can be viewed with `o/view`. Such viewer is a software device that draws and handles user interaction (eg. mouse and keyboard) and runs at a machine providing the screen, mouse and keyboard (both a PDA and a PC shown in figure 1). Communication between `o/mero` and `o/view` is intended to let `o/view` update its widget hierarchy according to the file hierarchy provided by `o/mero`, and to let the viewer notify `o/mero` of high-level events. Thus, latency and bandwidth requirements between `o/mero` and `o/view` are not high (eg., they do not exchange image rectangles as VNC would do).

Our player, editor, clock, load-meter, and other programs do not have to be split in two pieces, like in Protium [11], to distribute them. Yet we achieve a similar effect. This means that we do not need to create a per-application protocol to obtain separate viewers, and we do not have to program the same logic twice. Furthermore, because all the UI code is kept within the window system (the viewers, indeed), applications are more simple.

The (centralized) window system, `o/mero`, accepts commands to replicate widgets and place copies at different parts of the tree. Because of its centralized implementation, replication is easy. Just attaching different viewers to different parts of the file tree provided by `o/mero` suffices to provide a distributed UI service. Because we can implement viewers for any platform, heterogeneity is dealt with within the device, i.e., within `o/view`, and the application

and the rest of the system may remain unaware of it. We have not explored this, but we could go even further, and use audio devices to implement viewers for `o/mero`, using voice-based menus, for example.

As figure 1 shows, this is very powerful. In the figure, two different widgets (represented by directories in `o/mero`) have been pulled out of their original UI panels, and replicated in two different screens. We can handle individual widgets, not just entire application UIs. As you can see, users are now able to use the precious space of tiny PDA screens to put there only the controls they care about.

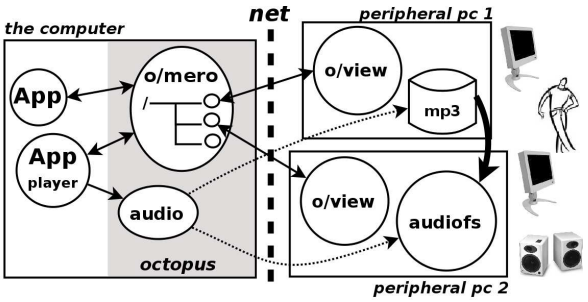

Figure 2: *Octopus working for a music player.*

How can our player play audio? It takes MP3 data and writes it to the audio device (ie. `/devs/audio/output`) The point is that the audio device of the computer might be indeed an MP3 player program running on the handheld. The audio device exports a file tree to the computer, what does not require a particular low latency.

It is important to notice that what we said for these two services is *applied to all system services* and devices, not just for UIs and audio. As of today, we have implementations for services that run on Plan 9 and Plan B and are imported by the Octopus. We are also using mices, keyboards, audio, and speech facilities attached to Linux machines in our current prototype for the Octopus. All the machines shown in the little image of figure 1 are controlled using a single keyboard and mouse, but we can also use any other mouse from any machine close to the user.

5. Copy devices

The centralization of control that guides the design of the Octopus implies a serious problem: in some cases, a data transfer may be necessary between two devices close to the user but both of them far away (measured in latency, or bandwidth) from the central computer. This is likely to happen often (eg., when reproducing a video media near the user at a player device also close to the user, while the computer is at other location).

The solution adopted in the Octopus is to include a *copy device* on each machine exporting any other device to the central computer. A copy device accepts copy requests for transferring remote files to a local destination (provided it has been granted the access rights and the address of the remote files). We have an initial implementation for this device but are still working on it.

Thus, in the Octopus, a player should call `copy` with two files to reproduce MP3 files. The first file is the MP3 file, the second file is the device file. The implementation of `copy` uses the copy device co-located with the target device to request a copy operation from the source file. In that way, data flows naturally from the source to the destination without involving the central computer in the data stream. The central computer is just controlling the devices (besides being the one executing the application).

6. Related Work

Protium [11] proposed to split applications into two pieces: the application proper and a viewer. That permits to use viewers for Protium applications at any other system. However, it requires a per-application protocol, and it is not clear if it requires reproducing the application logic twice (in the viewer and in the application). Unlike Protium, we apply that idea to devices and resources. For example, the UI and audio devices are split in the Octopus. Their interfaces are provided at the central computer, but their actual implementation remains at peripheral machines. On the other hand, our applications may be implemented as a single piece, unaware of this.

Internet suspend-resume [6] is similar in that it proposed using remote machines as terminal devices (hosts, indeed) for a single, central, per-user computer. However, it is not clear how to integrate multiple devices perhaps coming from different machines near-by the user in the computing environment as seen by the user and the computer. The Octopus can do it.

Systems like Globe [9], Ninja [3], Gaia [8], IWS [5] and One.World [4] rely heavily on middleware as the means to implement and distribute their services. Furthermore, programming applications for them is not trivial. The Octopus is a radical departure from their models (and from many other systems not mentioned here) in that it does not use middleware, and it does not require programming distributed applications, yet it provides the user with a distributed system that can be used as soon as devices nearby the user are

exported to the user's computer (perhaps by downloading a set of tiny user-level file servers from a web page, or by using an Inferno Web plug-in.).

Plan B [1] and Plan 9 [7] are direct ancestors of the Octopus. The Octopus is indeed a modified Plan B, which is actually a modified Plan 9. Unlike Plan 9 and Plan B, the Octopus does not require machines running Plan 9 or Plan B to let the user use them.

7. Discussion

The most important drawback is that the Octopus *wastes resources*, although that is indeed our intention, and it is deeply assumed by our design. A countermeasure is to try to exploit idle resources by making them available to the computer. Of course, most of the code to run user interfaces and to operate devices would be running at peripheral devices and computers, relieving the central computer from that task.

A failure in the central computer renders the whole computing system useless. This means that measures must be taken to make it highly available. On the other hand, users can drop any faulty device (not in the central computer) and replace it almost as easily as a pen is replaced by picking up a new one.

We believe that the main benefit from our design is *simplicity*. Because all the implementation is kept centralized, the system and the applications can be kept simple. For example, authentication is trivial, because the central computer includes a single authentication server, used to secure all user's devices. The same happens to other services, there is a single file system (that may use distributed storage), a single window system (that has multiple wigdet viewers at remote machines), etc. The system is consistent, because it is kept centralized. Despite this, it seems to be responsive when used from remote machines, because most user interaction happens locally, within the window system viewer, and because the high-level of abstraction used for system interfaces.

Another important benefit is *ubiquitous access* to the system, because most popular systems are to be considered resource providers for the Octopus. This includes web navigators as user interface devices.

A non negligible benefit is *fast boot* and suspend/resume, because the system will never shutdown. Only, external devices may be disconnected and reconnected. But both the system and applications will continue operation despite this fact.

Simplicity applies not only to programmers, but also to users. Today, when they use several computers, it is usual that they consider one of them as the main computer. Files created in other computers are usually transferred, manually, through the most rudimentary ways—sometimes as attachments to e-mails sent to themselves—, to the main computer. Users are forced to be aware of the heterogeneity and practical isolation of their computers when they are forced to move from a computer to another to use devices attached to a particular system, even though a network links all the computers. Worse yet, if not in the presence of what they consider as *their own computer*, they just can't (or won't) work. This natural tendency to centralize things suggests an easy transition to using the Octopus, as it is designed with centralization as its main design principle. The difference is that users would not have to think themselves about where are their files, devices or applications, and how to transfer the files to the main computer: the Octopus offers them the illusion that every application is launched and every file is stored on the main computer, transparently, even though users might be using hardware that is not close to their computer.

References

1. F. J. Ballesteros, E. Soriano, K. L. Algara and G. Guardiola, Plan B: An Operating System for Ubiquitous Computing Environments, *IEEE PerCom. Also http://lsub.org*, 2006.
2. S. Ghemawat, H. Gobioff and S. Leung, The Google File System, *19th ACM Symposium on Operating Systems Principles*, 2003.
3. S. D. Gribble, et al. The Ninja architecture for robust Internet-scale systems and services, *Computer Networks. Special issue on Pervasive Computing 35*, 4 (2000), .
4. R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad,, G. Borriello, S. Gribble and D. Wetherall, System Support for Pervasive Applications, *ACM Transactions on Computer System 22*, 4 (Nov 2004), .
5. B. Johanson, A. Fox and T. Winograd, The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms, *IEEE Pervasive Computing Magazine*, April 2002.
6. M. Kozuch, M. Satyanarayanan, T. Bressoud, C. Helfrich and S. Sinnamohideen, Seamless mobile computing on fixed infrastructure, *IEEE Computer*, 2004.
7. R. Pike, D. Presotto, K. Thompson and H. Trickey, Plan 9 from Bell Labs, *EUUG Newsletter 10*, 3 (Autumn 1990), 2-11.
8. M. Roman, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell and K. Narhstedt, GaiaOS: A middleware infrastructure to enable

active spaces, *IEEE Pervasive Computing Magazine*, 2002.

9. M. Steen, P. Homburg and A. S. Tanenbaum, Globe: A Wide-Area Distributed System., *IEEE Concurrency*, Jan-Mar 1999.

10. A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham and C. Yoshikawa, WebOS: Operating System Services For Wide Area Applications, *Proceedings of the 7th HPDC.*, 1998.

11. C. Young, L. YN, T. Szymanski, J. Reppy, R. Pike, G. Narlikar, S. Mullender and E. Grosse, Protium, an Infrastructure for Partitioned Applications, *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.