Traditional Systems can Work Well for Pervasive Applications A Case Study: Plan 9 from Bell Labs Becomes Ubiquitous.†

Francisco J. Ballesteros, Gorka Guardiola, Enrique Soriano, and Katia Leal
Laboratorio de Sistemas — Universidad Rey Juan Carlos
Madrid, Spain.
`http://lsub.org/who`

ABSTRACT

There is a huge effort in ongoing research on new middleware platforms and new distributed services to support ubiquitous environments and pervasive applications. Most research projects mention concrete applications or demonstrators used to back up the claimed need for a new particular service, piece of software, or middleware layer. However, we have found that in many cases we could easily build such applications by relying on services already provided by the system we use daily, Plan 9 from Bell Labs. This paper explores how far can we go with a traditional system to support an ubiquitous environment, with no need for new services. We describe how we used Plan 9 to make our environment become ubiquitous. We describe as well what limits we found, and what technology could be used to overcome them.

1. Introduction

Since Mark Weiser advocated for a pervasive computing era [22] the research on pervasive computing has been always increasing, and still is. Part of the work published addresses the issue of how to use new services in a clever way to reach the technology level necessary to implement the scenarios being considered. Other efforts focus on implementing the new services that are necessary to provide required environment. Both things are necessary to build pervasive environments. However, there is also much work done on new system architectures and how to integrate services on them, eg. [4, 6, 8, 15, 17]; we argue that in many cases using existing systems may be a good alternative.

Since as researchers and engineers we are always eager to develop new building blocks, all of us have a considerable tendency towards developing new architectures and services. This might make us overlook many cases when the tools readily available suffice to reach the new goals, and there is no need for the new architecture or service. We asked ourselves if that was the case, at least in part, with pervasive computing environments. Therefore, we decided to try to build a pervasive environment out of the tools we had, and we have found we mostly can. A demonstration of the resulting environment can be found at http://lsub.org.

That is not to say that no new service is needed for pervasive environments, nor to say that new application frameworks would not permit a better way of doing things. That is just to say that many times we can do well enough, if not better, with the existing set of tools.

Our main tool is Plan 9 from Bell Labs [14], an Operating System built on the 90s, that runs on many platforms going from Intel based PCs to iPAQ Pocket PCs. Briefly described, Plan 9 is distributed system built along the idea of exporting all resources as files. Since the distributed file system technology is well-known and well understood, their authors decided to model (almost) all system resources as files and permit all files to be used through the network. In that way, the system becomes distributed. By using per-application name spaces, and not one per machine like in UNIX, Plan 9 applications can customize their environment by importing from the network the set of resources (ie., of files) they need.

In this paper we describe what we have done to Plan 9 to turn it into a pervasive computing platform,

† This work supported in part by spanish MCYT TIC-2001-1586-C03-01 and URJC PPR-2003-40.

without using middleware, introducing new abstractions into the system, or relying on external services. Our approach has been to integrate into the system the new devices necessary provide control over the physical environment, and to inspect it. We have also experimented with the extraction of context information [9] and have used our platform to build several pervasive applications that go from (physical) environment automation down to a new `who` utility. Some of our applications are also context-aware. A demonstration can be seen at http://lsub.org.

The resulting environment is being used daily at the distributed environment that integrates our offices and meeting rooms and has become a useful and convenient platform. It includes control over cheap computer controlled power switches, lights, motion detectors, desktop computers, mobile devices like laptops and Pocket PCs, and touch screens among others.

Moreover, we have built our environment in such a way that it interoperates well with systems like UNIX, Windows, and Symbian. This permits us to borrow services from other systems and to use our services from other platforms. Interoperability did not require new technology yet resulted straightforward to achieve. Regarding scale, the system has been used only by our group, but we firmly believe that it scales well, as we justify later. As we will say, we have found some problems that need further work. We are working on fixing them by applying ideas from Plan B [2] to our environment.

In what follows, section 2 briefly describes Plan 9. Section 3 describes the environment we built. Section 4 discusses some important issues like scalability, naming, interoperability, type checking and protection. Section 5 shows some applications developed by us that use the new environment. In section 6 we mention some problems yet to be solved. The lessons we learned are shown in section 7. Section 8 compares our approach to some related work. We conclude and state the plans for future work in section 9.

2. Plan 9 from Bell Labs

The operating system we use for daily work is Plan 9 from Bell Labs [14]. It is a distributed system developed in the late 80s and early 90s that has been on production since then. The system is built upon these ideas:

- **Everything is a file.** Almost all system resources are provided as if they were files. For example, processes are killed and debugged by using files; The audio device is represented by a couple of files, one that represents the volume level, and one that represents the output device for audio data. The same applies to windows, network connections, etc.
- **All files are accessible remotely.** The system speaks a network file system protocol, 9P [14], to operate on remote files. For the system user, there is no difference between local and remote files.
- **Each application has its own name space.** Each process can customize its name space (eg. what in UNIX would be its mount table) according to its needs. There is a per-process mount table that permits a process to select which resources are mounted on which names [13].

Note that the combination of the first two principles provides a simple to use distributed system. Since all resources are exported by means of files, and all of them can be used remotely, all resources can be used from somewhere else in the network. The last principle provides for customizability of the environment.

3. The URJC Systems Lab's Smart Space

Smart spaces [10, 17] integrate computer facilities into the physical environment, allowing the system to operate on the environment. They aim at supporting human activities within the space. To do so, new sensing and actuating devices are installed on the physical environment and integrated into the smart space middleware[1]. For example, a smart space is supposed to allow the computers to operate on power sources to switch devices on and off, as well as to be able to detect whether there are users in the space or not. The middleware layer distributes the services provided by the new hardware, and concocts its own services. As examples of such services, these systems use to include the ability to start applications on different computers and permit the use of different I/O devices for a given application depending on the environment circumstances. Besides, the system has to deal with issues important for applications, like mobility of users and devices, managing context, notifying events, protecting the distributed environment, etc.

[1] Such systems are usually implemented by means of a middleware layer.

The Systems Lab's smart space is a prototype environment that comprises most of our offices and several common spaces like for example our mail and meeting rooms. Figure 1 depicts our environment, which includes devices like touch screens and X10 controlled power switches used to control lights and power sources for computers. Some of the rooms include motion detectors, and the mail room has a camera that is used to detect the presence of (real world) mail in the mail boxes. Using these devices, we have integrated our computing system and our physical environment. For example, some of the users that use to listen music at high volume levels, have automated their environment to get their volume output level lowered when they are being visited. In the same way, some arrange for their audio devices to get muted when nobody is listening. Lights and other devices like touch screens have been automated in the same way.

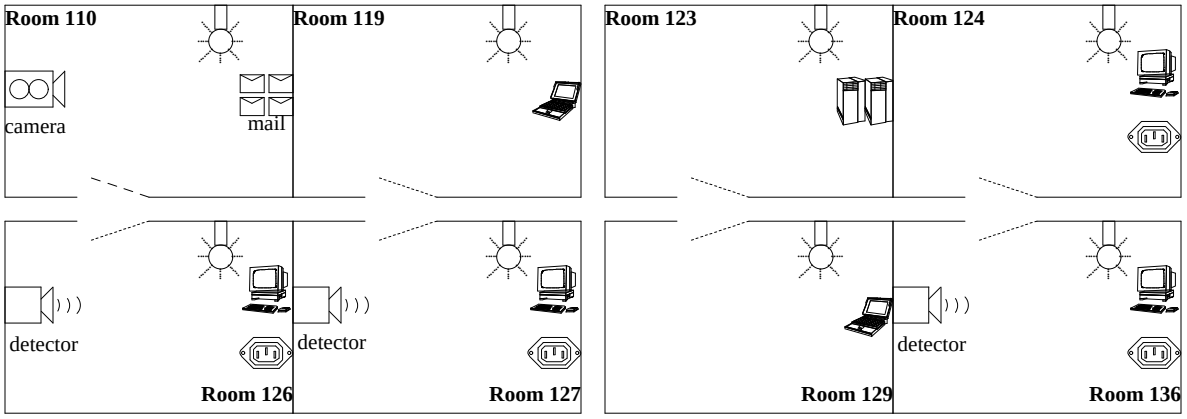

Figure 1: *The Systems Lab Ubiquitous Environment*

Unlike other smart environments, ours is not at a separate laboratory, nor is being used just for meetings. Our environment is in production at our offices and common spaces and we are using it daily. The same computing system that manages our environment is being used at our laptops, at home, to give lectures, to do research and, of course, to implement further services. The system is made of machines running Plan 9, Windows, and Linux. The machines go from servers and desktop systems down to iPAQ pocket PCs[2]. We do not use custom made hardware, all the devices in use are available from the industry.

3.1. Integrating the Physical Environment and Plan 9: The X10 service

To control power sources and lights, we use X10 (see www.x10.org). In figure 1, the power outlets represent X10 devices. The switches powering terminals (desktop PCs) accept *switch on/off* commands. The ones connected to lights also accept light attenuation requests. X10 CM11 controllers, attached to server machines at room 123, permit to send commands to other X10 devices, and to obtain their status. Besides, there are several X10 motion detectors like the one shown on the left of room 126.

For power switching all we need is to be able to check the status of each power source, and to be able to change it. For motion detection we need just to be able to check the status of each sensor. Therefore, although the two services considered are different (power switching and presence detection), their behavior is similar enough that we use a single program to integrate them into the system.

Using Plan 9, it was obvious for us that a good way to provide the X10 service is by means of a file system. The new file system services a flat directory with a (synthesized on demand) file per X10 device. A device status can be accessed by reading its file, and updated by writing to it. This means that both users and programmers can rely on the well-understood file system interface to operate the new service. Should we have provided the service by using distributed components or a middleware layer, accessing the service would have been more

[2] We are also in the process of integrating mobile phones as general purpose terminals for our environment.

complicated, and interoperability would be worse. Most machines, if not all, know how to access files remotely; But not all machines know how to access a service through a particular middleware or component technology.

The interface for the resulting service is very simple. Each file appears to contain either the string `on` or `off` depending on the status of the corresponding device. For a motion detector, we assume that `on` means a positive detection, and `off` a negative one. A device is switched on by writing the string `on` to its file. It is switched off by writing `off`, instead.

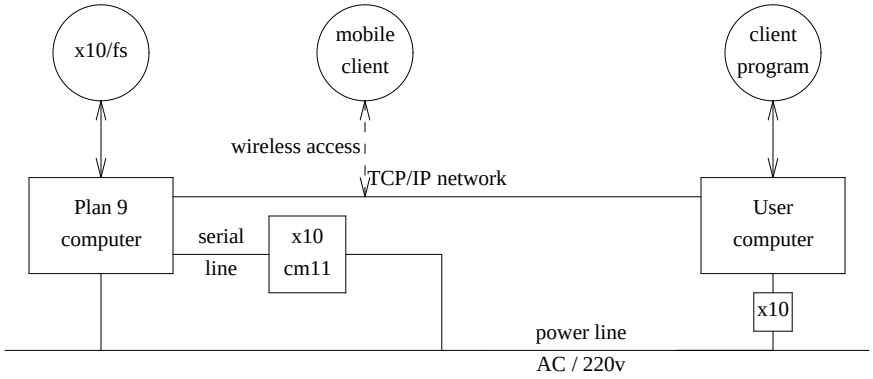

Figure 2: *The X10 file system: Integrating X10 devices into the network*

The overall architecture is shown in figure 2. A program called `x10/fs` speaks the Plan 9 remote file system protocol, 9P, to export the file system interface for a CM11 controller. Once started, users can mount the file system on a place of their choosing from any place in the network. User programs, and new applications, can use the files to check or set the status of the power source devices and the motion detectors. A mobile device with TCP/IP access can mount the file system and access the X10 services. In the same way, a client program from a X10 powered machine can mount the file system. Both clients could issue commands and check the status for X10 devices.

3.2. Context Handling

A full discussion of our context architecture is done somewhere else [9]. However, we describe here how context handling can be also performed on a traditional system.

The context-awareness "framework" we use is simply a set of directories. Our main file server includes a series of directories where context information is to be placed, so that no other file system needs to be mounted to access context. Nevertheless, we also have small file systems to keep context for mobile devices and people. They are similar to ram based file systems used for temporary storage. Such file systems are handy while outside of our smart space—ie. while disconnected at home or while we are in the subway. Figure 3 shows a typical context hierarchy.

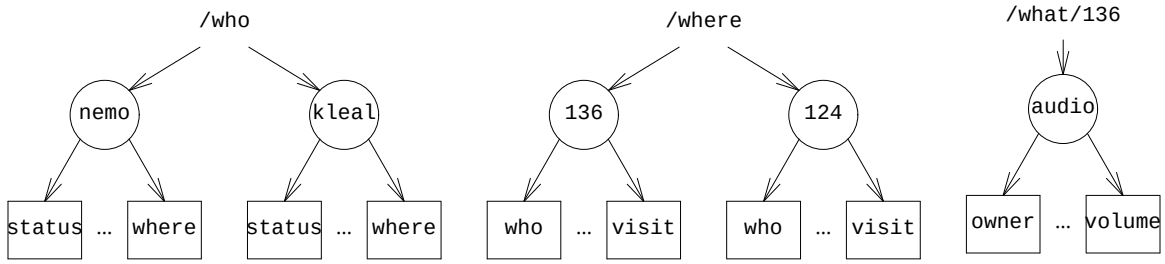

Figure 3: *Our context hierarchy. Used to update and to use context information.*

Users have a directory (one for each user) for their relevant context information. Each piece of context is represented by a file in that directory. For example, `where` is a file that contains the last known location for the

user, and `status` is a file that contains a descriptive string about the user status (eg. `busy` or `idle`). Each place has also a directory that contain its context information. As examples of context information for places we can mention the file `who`, which contains one line per user known to be at the space, and also `visit`, which contains either `yes` or `no` depending on the answer to *Are there humans present in the space other than its owner(s)?* Finally, things (including devices and services) also have their own context directory. For example, the context directory for an audio device includes a `owner` file containing the device's owner; there is also a `volume` file containing the device output volume level desired by the owner.

The context information stored is not assumed to be accurate, it is as accurate as the tools used to extract it from somewhere else in the system. The set of tools used to extract, merge, and use context information is still growing. It includes tiny shell scripts as well as more complex C programs.

Users, and space administrators, are free to run whichever tools they see that fit to extract and use context. The different tools work together using the file system to exchange information. This works very much like the UNIX environment did time ago, by combining simple programs to perform complex tasks. Therefore, we do not have to use one solution for all the problems, and users can customize how the system extracts and uses context on their behalf.

To make an example, the owner of room 136 prefers to say that he is the one using that room as soon as the motion detector focused on his desktop detects movement. This works for him because he is the room owner and has a key[3]. As another example, `fcompose` is a tool developed by us that evaluates a set of rules to determine the value for a piece of context. `Fcompose` takes a set of files (which might represent context information as well) and *merges* them according to a set of rules. The output text is usually copied to another file to store the resulting context information. Yet another example is using RFIDs or Crickets [16] as the source of information to update `who` files for places and `where` files for persons.

4. How Does This Approach Work?

Before discussing some concrete applications used in our environment, it is better to show how our approach addresses different issues like interoperability, naming, type checking, scalability, and protection. Although we will be using our X10 service to illustrate the discussion with some examples, what is said below also applies to other services.

4.1. Interoperability

We do not use any standard middleware technology for interoperability, yet we are able to interoperate with all the machines we know about. This includes Windows, Linux, Plan 9, and Symbian running on Intel PCs, PowerPCs, Strong Arm Pocket PCs, and mobile phones.

The new services for pervasive computing provide file interfaces to actuate on them and to check their status. They are exported through a particular network file system protocol, and it might seem that it is necessary to speak that protocol to access them. However, what is important is that files can be accessed through the protocol *spoken by the client* machine. Since, besides 9P, our servers speak CIFS [11], NFS, and HTTP, most machines can operate on the files used to provide the new services. Moreover, by using intermediate machines that speak protocols not supported by Plan 9 (eg. bluetooth file sharing [3]), we can interoperate with even more machines. Note how this approach permits the immediate integration of other platforms, which can now use the new services.

We have to say that, according to our experience, platforms like Java have more problems to interoperate (due to version and library problems) than we have found using a network file system. Small mismatches between different network file system protocols (eg. different metadata, or a different set of file operations) are usually solved or worked around by the gateway machines. But this is not usually true with distributed middleware components. In fact, it is usual that many middleware systems end up relying on XML or other textual descriptions just to be able to interoperate. For example, see any of [4, 6, 20]. In our case, the file system together with a set of conventions is giving that interoperability for free.

[3] And because he does not care if the cleaning service triggers a false location for him!

4.2. Naming

To name devices, we use file names. Instead of using the naming scheme of the technology used, the names used are meaningful to humans. For example, X10 uses the name `a1` for the device that powers the light at room 136 (house code `a` and device id `1`). However, that device is shown as a file whose name is `pwr:136light`, which self-explanatory for our users. We use names starting with `pwr` for power-source switching devices and names starting with `who` for motion detection devices. Each name also includes the room number for the device and the name of the element controlled by the device. The X10 file server imposes this naming convention, so that users do not have to care. Actually, users are not able to create files at an X10 file tree. Other file servers for related services impose their own naming scheme too.

By following this simple naming convention, we simplify tasks that would require more complex technology. For example, X10 devices do not have location information yet a user can find out which X10 devices are at room 136, or which ones of the devices at room 136 are switched off:

```
; 9fs x10 # mounts the X10 file servers
post...
; ls -l /n/x10/*136*
--rw-r----- t 0 nemo nemo 0 Mar 22 16:14 /n/x10/pwr:136term
--rw-r--r-- t 0 nemo nemo 0 Mar 22 16:14 /n/x10/pwr:136light
--r--r--r-- t 0 nemo nemo 0 Mar 22 16:14 /n/x10/who:136
; grep off /n/x10/*136*
/n/x10/pwr:136light: off
```

On its own, what has been said would not suffice to organize the set of devices found in a typical department. However, we must remember that client applications can mount the servers on a place of their choosing. Therefore, the name space is a hierarchy where devices and services are integrated as any other file used by the client. In our case, the convention is to mount X10 devices at `/n/x10`, context for persons at `/who`, context for places at `/where`, and context for things at `/what`. But note how any other organizational convention could have been applied as easily. We follow the Plan 9 approach for managing name spaces. The system administrator establishes common name spaces, and users customize their name spaces using their profiles. Finally, some concrete applications modify their name spaces when they need to.

4.3. Type checking

There is no type checking in the file system (other than distinguishing between files and directories). The files that are used to provide a service (like the X10 ones) do not type check what is being written to a file. There is a single data type, the string, used to perform all the data exchanges. This is good for portability, since strings use to have the same format at most machines and they are handled well by the existing user interfaces. Furthermore, debugging is greatly simplified, since the strings are well understood by humans[4].

Although this might suggest that we have a problem, and that all of our servers admit any request (meaningful or not), that is not the case. We now discuss how the two most important problems caused by the lack of a type checking system are not a problem in practice.

1 **Errors due to erroneous data** are not a problem for us in practice. When users write a wrong string to operate on a device or service, they usually get a “bad request” error message and no operation is actually performed. Since different devices and services have different operations, it is very unusual to see a request for a service work on a different one. In fact, Plan 9 relies on this for all the services it provides, and it is a very reliable and convenient system to use.

2 **Exchanging complex data** is not a problem either. Complex data is modeled by a hierarchy of files, and not by a file on its own. This means that to exchange complex data “types”, we exchange file hierarchies. The directories in the hierarchy give us most of the functionality of the XML tags: They impose a structure on the data and they name data sections.

An example can be found in our GUI service, that relies on file hierarchies to exchange, move, and combine

[4] Even better understood than XML files, that require some parsing before the human could grasp where the interesting data is.

graphical user interfaces between different types of machines and displays. Although a discussion of this service is outside the scope of this paper, we can say that it represents graphical widgets by files, and compound widgets (eg. menus) by directories. A typical user interface is built by creating a file hierarchy on a GUI file server. The file server creates widgets and implements them.

4.4. Scalability

Our installation does not have a large scale, although it is comparable in size to other ones reported in the literature [10]. We are convinced that our scheme scales well. The reason is that there is no central-server in charge of all the devices. Besides, none of the mechanisms used to access the services (eg. naming) are centralized. The key issue is that users customize their environments by mounting whichever servers they want on whichever places they choose. The overal scheme is very similar to that of the web, which admittedly scales well.

The system scales because the mechanism that must support the growth of scale is the file system protocol used to access the resources. We already know that this mechanism scales well enough to service at least the entire department. As a result, our servers can be kept small, and clients can decide not to mount at once all the servers available—if they are not powerful enough to do so.

Consider the X10 service as an example. Each X10 CM11 controller manages a handful of devices for several rooms and has its own X10 file system server. The file system server is just a process on the machine where the CM11 is attached. A user who wants to see all X10 devices at a single place would mount all the known file servers at a single directory. The directory contents would appear to be the union of all the ones serviced by the servers. But another user might create an empty directory per building corridor and mount there the corresponding servers, to see which building corridors have which X10 devices. Needless to say, this can be automated with tiny shell scripts provided by the system administrator—when not automated by using RFID tags attached to the X10 devices to determine where to mount them according to their location.

4.5. Protection

There has to be a protection and security management mechanism for the real world as well as for the computer system view of such world. Having just one for the computers is not enough. For example, figure 1 shows how several X10 controlled power sources are used for terminals. The power sources, once accessible by the computer through X10 devices (or any other means), should be owned by the users who own the terminals powered by them. Otherwise, a user might make a mistake and switch off a power source for another user's terminal.

Since we use a file system to export the X10 devices to the computer system, we have protection mostly solved. Authentication is performed when mounting the X10 file system as it happens to our main file systems. For authorization and access control, we rely on file permissions: file ownership is assigned to represent device ownership, and file permissions are used to perform access control. This solution is both simple and effective. If room 136 of figure 1 is Nemo's office, and room 124 is Katia's office, a list of the relevant X10 files can be as follows.

```
; ls -l /n/x10/*136* /n/x10/*124*
--rw-r----- t 0 nemo nemo 0 Mar 22 16:14 /n/x10/pwr:136term
--rw-r--r-- t 0 nemo nemo 0 Mar 22 16:14 /n/x10/pwr:136light
--r--r--r-- t 0 nemo nemo 0 Mar 22 16:14 /n/x10/who:136
--rw-r----- t 0 kleal kleal 0 Mar 22 16:14 /n/x10/pwr:124term
--rw-r--r-- t 0 kleal kleal 0 Mar 22 16:14 /n/x10/pwr:124light
--r--r----- t 0 kleal kleal 0 Mar 22 16:14 /n/x10/who:124
```

It can be seen how Nemo allows any user to read his detector status, but only him can switch on/off his light and terminal. Should Katia want to disallow for others to know if her motion detector detects movement in her room, she could execute:

```
chmod o-r /n/x10/who:124
```

In the example above it is worth noting how a simple file list command could be used, and no special browser was needed. In the same way, any Windows machine is able to use its file explorer to show the devices,

without any special plug-in for our new services.

Device ownership is assigned statically for devices located in a room that is assigned statically to a user. For devices located in shared spaces, ownership is assigned dynamically. In our case, the power-source owner (in a shared room or space) depends on who is using the terminals. Devices like light power sources in shared rooms are assigned to the group of users entitled to use the room, since it is not necessary to own a terminal just to control the lights.

5. Applications developed

Figure 4 shows the interface of some of the first tools we developed. The panel is simply a user interface that shows the X10 files available at `/n/x10`, using different colors according to the strings they contain. A mouse click on a file would write `off` if the file contained `on` and vice-versa. The slider on the right is a volume control that writes to `/what/.../audio/volume` the volume level selected by the user. The program `x10/power` is an automation tool whose log entries are shown near the bottom of the figure. This program limits the maximum volume level when the `/where/.../visit` file for the place contains `yes` (to avoid disturbing while visiting). It also mutes the audio output when the `/where/.../who` file is empty (nobody is listening). Another program updates the files mentioned according to the contents of the X10 movement detectors (accessing such devices by reading its files).

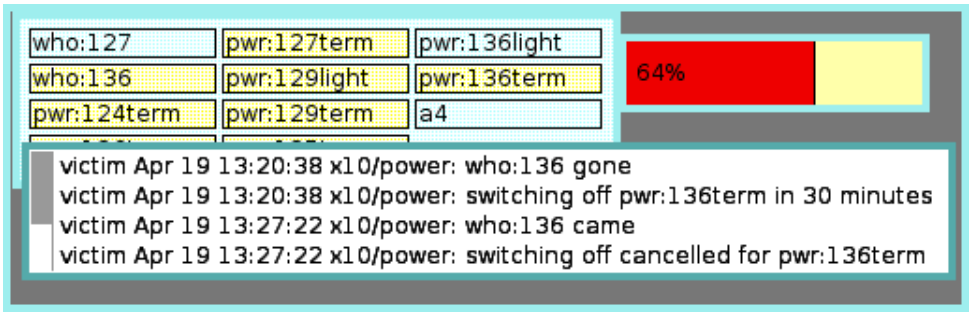

Figure 4: *Programs using the location and power infrastructures*

This is just a particular setup, but note how it would be quite simple to use different programs to arrange for a different one.

Figure 5: *Programs using the* `/who` *service*

Figure 5 shows the interface for a `who` utility that shows who is in the space (no matter at which room). In effect, `who` has brought back the venerable UNIX tool of the same name. However, our tool works for the physical space and does not need the user to be logged in nor to use any particular operating system. The flexibility of our approach can be seen by considering what would be needed to change our particular setup. Since our rooms are close enough, using a single `who` window works well for us. Consider now that things change and we end up with several groups of rooms, far apart. We can simply run several instances of `who`, mounting for each instance the set of files for places that are close to it. This leads to several `who` windows, one for each

group of rooms. Users who prefer to make any other arrangement can do so. That is to say that our approach leads to very modular services. As another example, the web pages at `http://lsub.org/who`, and `http://lsub.org/where` export a read-only version for part of `who` and `where`.

The location and status information kept at `/who` comes from many different places. For example, we are combining X10 sensors, status for users in the MSN Instant Messenger, pings to user laptops, and various output files supplied by various users. Since the files under `/who/`*user* belong to each *user*, we allow our users to update their status files in any way they can imagine. Users can either rely on system services, eg. X10, or implement any heuristic that works for them. Some of them do so on Linux, others on Plan 9; some use shell scripts, some use C programs; etc. For example, some of us update our MSN status according to the motion detectors in our rooms. But there are others who do just the opposite, and use their MSN status as the source. In few words, this approach has proven to be extremely flexible and easy to use.

Finally, the shell window on the right of figure 5 shows how a simple command can be used to see which users are away or to determine the current location for user `paurea`. As in the previous example, the tools are quite simple and rely just on the files that are exporting the service.

6. Problems to be solved

The main problem we have is adaptation to changes in the environment. In particular, Plan 9 keeps network connections between clients and servers, and the servers are state-full. The main problem we had was to maintain the list of things that clients can (or must) mount, since the list changes depending on the environment. For example, which X10 servers are the ones that a client machine should mount? Addressing this problem requires a resource discovery service and some means to specify which preferences each user or application has about resources. Fortunately, Plan B (a research system that we built) addresses these problems. You can refer to [1] both for a description of Plan B and to see how we have incorporated its mechanisms into Plan 9 to get a system that adapts to changes.

Another problem is that the approach shown in this paper still requires some centrally administered servers. The problem is not administrative, since we have to administer just one of them: others are configured and booted from that one. However, we want to use our services (eg. X10 devices) from our mobile devices even when our main servers are down. The problem we face is that even though we can mount any service from anywhere without the help of any central server, we still rely on our authentication servers to secure the remote access. To solve this problem, we are implementing a peer-to-peer, human centered, approach to security [18]. This will provide us secure access to services provided by other machines.

A related problem is that the servers with our X10 controllers may fail. But we think this problem can be addressed by embedding our device file servers into the devices of interest, and by replicating the controllers. Nevertheless, more experimentation is needed.

7. Lessons learned

One lesson that we learned is that providing software for a new service is not enough to claim that the service can be used. For example, the vendor software for our X10 devices includes middleware that allows to see the status of the devices, to change their status, and to automate a minimal subset of tasks (like switching on a device when the motion detector reports activity). However, after installing such software, we found ourselves using just the remote control to switch on and off the devices. The technology of the new service was wasted. Nobody was using the devices through the computers, and many X10 controllers were sitting on desks. However, things changed when the service was exported as files: Any user could easily access and program the service, no matter the programming language, scripting tool, or operating system used.

But perhaps the biggest lesson we learned is one we already knew (but somehow forgot) from UNIX. It is the combination of multiple (usually tiny) applications what makes the services useful. Most of the applications are in fact made by the users and were not envisioned when the services were installed. Existing tools are combined by means of the shell and other system facilities, to yield new tools. This is what made UNIX powerful, when compared to other systems of the epoch. We are just following this approach for the new environments that we have now. What makes the service really useful is the ability to integrate it with the rest of the system without forcing the user to employ complex programs or tools. For example, before using our context file system, one author used the X10 motion sensor to mute his terminal audio output when no one was

listening. The script is shown here; it is simple, yet performs a useful job.

```
#!/bin/rc
9fs x10             # mount the X10 file system(s)
while(1){
          sleep 60
          if (~ ‘{cat /n/x10/who:136} "on")        # file contents == "on" ?
                    echo "audio out 60" >/dev/volume
          if not
                    echo "audio out 0" >/dev/volume
}
```

Note how, by accessing both audio and X10 through files, it becomes really easy to combine them. Regarding X10, we have built a generic information and actuation service at the system level. Each person can employ it for whatever purpose can be imagined, there is no need to modify or program a complex application to do so. The C program that switches on the terminal when the user enters the room, and switches it off when the user leaves for more than 30 minutes is 217 lines of C code. This program has nothing to do with the script shown above, and both programs integrate seamlessly with the rest of the computing system. The program is “so big” because it also knows how to handle other devices not mentioned here.

Middleware is good to experiment, but it is not so good to provide system services. We have found that our applications are ready to use the new services because the system provides them. As our previous experience with middleware systems shows, that is not the case when the services are provided through middleware; because applications must know how to use the middleware. For example, we were amused to find out that by exporting our X10 file system through CIFS, the Windows notepad could be used to check and change the status of X10 devices. Of course the notepad was developed without that usage in mind! This would not happen if the service had been exported through complex J2SE and XML based interfaces.

A consequence of this is that since all the applications are ready to use the new service, the service is actually used daily. Therefore, experience is gained quickier than it would be otherwise.

We have also found that it is important for the system to provide just the mechanisms and not the policies. For the services discussed in the paper we provided the mechanisms, the policies were implemented by user programs. Different policies were used by different users in a natural way, without having to change the software that provided the service.

Another lesson learned is that protection in the physical environment can not be that of a traditional system. For example, if a room is entitled to a person, all the devices in there must be owned by such person. However, the devices that can disrupt operation of computers (eg. power sources), must be assigned to whoever is using the computers. In few words, the protection scheme used within the computers must borrow properties from the real world.

X10 is quite unreliable and is specially bad regarding protection, since there is no security issue considered in the protocol. However, our scheme works well and is reasonably safe. But note that any casual user may plug an X10 controller in a wall socket and use a remote control to issue any X10 command to any X10 device connected through the same power line. Fixing this would require using different controllers or installing X10 devices that disrupt X10 signals to isolate power lines.

The automated switching on/off of the terminals has shown to us that it is useful to introduce hysteresis on dangerous operations. For example, switching on a terminal can be performed immediately. Switching it off, on the other hand, requires a safety delay between the request and the actual power-off. Such delay allows for the user to quickly undo the mistake, and allows any safety counter-measure to cancel the dangerous operation. As another example, a user can issue a lights-off command for a light in the stairs, but a motion detector should have time to override the command when it detects people going downstairs.

This could be done by introducing the safety checks on each application. But the hysteresis permits the implementation of simple watchers that assert safety conditions independently of the applications (eg. if there is motion and there is no light in the stairs, keep the lights on).

Regarding system administration, it is crucial that servers deployed can run unattended. Otherwise, they run quickly out of service due to human maintenance failures. We achieved this by banning local storage and local administration commands on the servers where the X10 controllers are installed. All of them boot from the

network, and the file servers use a single configuration file that is centrally administered.

8. Related work

The main difference between our approach and other ones is that we use existing system services as tools to provide the new services. Due to the lack of space, we mention here only the most representative related work, and focus on the main differences.

There are many other systems, like Ninja [7], Gaia [17], Globe [19], One.World [8], etc. that rely heavily on middleware as the means to implement and distribute their services. They do not provide a complete computing environment since their applications require services of the native system underlying their middleware. Our Plan 9 system is a complete and self-contained one. Another difference is that systems like Gaia rely on CORBA-based middleware and use to supply an scripting language, eg. [17], to automate the environment. Gaia users need to have at least one of both tools installed on their systems, ours do not.

Unlike in middleware based approaches, ours permits a native Windows or Symbian application to access the new services just by using the file system interface. Middleware is good to provide add-ons that cannot be provided otherwise, but as we have shown, that is not the case for the services discussed in this paper. Furthermore, middleware systems introduce more complexity, and this has an impact in battery, processor, and memory consumption that we plan to study soon.

Another big difference regarding middleware based systems is that we use well-known and well-understood distributed file system technology. An important consequence is that we interoperate with any system able to exchange or to remotely access files. Unlike the systems mentioned, we do not require Java nor any other platform in mobile phones, yet we are able to use the system from them.

Many systems improved distributed file systems to add features like multimedia services [5], disconnected operation [12], etc. None of them tried to use such interfaces as the primary interface for all the system services.

There is plenty of work about how to use XML and related markup languages to exchange data and support interoperation. See for example [6, 20]. The main difference between our work and them is that we use text based interfaces from the beginning. Our hierarchies are provided by the file system, not by the language tags. Furthermore, they usually focus on how to adapt one kind of data to another, and we are focusing instead on how to export and use the new services required for a pervasive computing environment.

WebOS [21] is close to our approach in that they tried to use a file system, the Web, to provide all necessary system services. However, their system is designed for large scale and not for a departmental service. It is also unclear what is their implementation status and how they would allow to program distributed applications. We could have used a web based interface. However, that does not solve problems like authentication, access control, and synchronization. Using a file system interface solves all of them.

Last but not least, one point that shows the difference between our system and related approaches is that ours is both our research platform and the system we use for daily work. Other systems are either used to carry out the daily tasks, or as a research platform. As far as we know, they are not able to serve for both tasks. In [1] we show a more in depth description of our system, that can better support this claim.

9. Conclusions and future work

Files are powerful, specifically when they are used for devices and not for data on a disk. Tiny programs each one performing a single job well, together with means to combine them, are more powerful than big software frameworks. They are simple to implement, easy to use, and need no further system services. Maybe we should reconsider the UNIX lesson instead of relying on complex middlewares exhibiting XML, P2P, and other technologies in the main stream.

We have just completed the implementation of an hybrid between Plan B and Plan 9, that we confusingly call Plan B 3rd edition [1]. This new prototype, as we said, addresses the problems of the approach described in this paper. In the near future we will continue using it, to learn from the experience. We are also installing indoor location systems and further devices. By following our approach, we are sure that all such services will be easy to integrate into the system, and easy to use. We also plan to measure the difference in battery, memory, and memory consumption between our approach and a representative one for middleware based systems.

References

1. F. J. Ballesteros, K. L. Algara, G. G. Muzquiz and E. Soriano, The Design and Implementation of Plan B 3rd edition. A dynamic distributed computing environment., *GSYC-Tech. Rep.-2004-05*, 2004.
2. F. J. Ballesteros, G. G. Muzquiz, 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. Bluetooth/SIG, Bluetooth Specification Documents, *Available at www.bluetooth.org/spec*, 2004.
4. G. Chen, Solar: Building a context fusion network for pervasive computing, *Ph. D. Dissertation. Dept. of CS. Dartmouth College.*, Aug. 2004.
5. S. Childs, Filing system interfaces to support distributed multimedia applications, *Eighth ACM SIGOPS European Workshop. Support for Composing Distributed Applications.*, 1998.
6. D. Garlan, D. P. Siewiorek, A. Smailagic and P. Steenkiste, Project Aura: Distraction-Free Ubiquitous Computing, *IEEE Pervasive Computing special issue on Integrated Pervasive Computing Environments 1*, 2 (April-June 2002), 22-31.
7. 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 Internet-scale systems and services, *Computer Networks. Special issue on Pervasive Computing 35*, 4 (2000), .
8. R. Grimm and B. Bershad, Future directions: System Support for Pervasive Applications, *Proceedings of FuDiCo 2002*, June 2002.
9. G. Guardiola, CUROCO: A distributed architecture for the dynamic generation, composition, and use of context., *First Intl. Middleware Doctoral Symp.*, 2004.
10. B. Johanson, A. Fox and T. Winograd, The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms, *IEEE Pervasive Computing Magazine*, April 2002.
11. P. Leach and D. Perry, CIFS; A common Internet System, *Microsoft Interactive Developer*, Nov. 1996.
12. B. Noble, M. Satyanarayanan, D. Narayanan, T. J.E., J. Flinn and K. Walker., Agile Application-Aware Adaptation for Mobility, *Proceedings of the 16th ACM Symp. on Operating System Prin.*, 1997.
13. R. Pike, D. Presotto, K. Thompson, H. Trickey and P. Winterbottom, The Use of Name Spaces in Plan 9, *Operating Systems Review 25*, 2 (April 1993.), .
14. R. Pike, D. Presotto, K. Thompson and H. Trickey, Plan 9 from Bell Labs, *EUUG Newsletter 10*, 3 (Autumn 1990), 2-11.
15. S. R. Ponnekanti, B. Johanson, E. Kiciman and A. Fox, Portability, Extensibility, and Robustness in iROS, *Proc. IEEE International Conference on Pervasive Computing and Communications. Percom 2003*, 2003.
16. N. B. Priyantha, A. Chakraborty and H. Balakrishnan, The Cricket Location-Support system., *Proc. 6th ACM MOBICOM*, Aug. 2000.
17. M. Roman, C. K. Hess, R. Cerqueira, K. Narhstedt and R. H. Campbell, Gaia: A middleware infrastructure to enable active spaces, *Technical Report UIUCDCS-R-20022265. University of Illinois at UrbanaChampaign*, 2002.
18. E. S. Salvador, F. J. Ballesteros, K. L. Algara and G. Guardiola, A Human Centered Security Protocol for Ubiquitous Environments, *Submitted for publication. Also at http://lsub.org.*, 2004.
19. M. Steen, P. Homburg and A. S. Tanenbaum, Globe: A Wide-Area Distributed System., *IEEE Concurrency*, Jan-Mar 1999.
20. uPnP-Forum., Understanding uPnP., *www.upnp.org*, 2004.
21. 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 Seventh Symposium on High Performance Distributed Computing*, 1998.
22. M. Weiser, The computer for the 21st century, *Scientific American*, September 1991, 94-104.