Plan B: Using Files instead of Middleware Abstractions for Pervasive Computing Environments†

Francisco J. Ballesteros, Enrique Soriano‡, Gorka Guardiola, Katia Leal
2/20/2006
Laboratorio de Sistemas — Universidad Rey Juan Carlos
`http://lsub.org/who`
Madrid, Spain.

ABSTRACT

Most approaches to handle ubiquitous environments use middleware to integrate different systems. Their motivation is usually interoperability and providing new abstractions better suited to new services for pervasive computing. A drawback of this approach is that general purpose tools cease to work for the new abstractions, which makes things worse for programmers because new tools must be developed even when old tools could work. Another drawback is that interoperability now requires deployment of the middleware considered, or code shipping. We believe that the traditional file system interface, combined with a sensible design of interfaces for devices and services, suffices both to make machines interoperate well and to provide the new abstractions and services required by ubiquitous environments. Our approach is to, first, put *all* individual resources in the network using distributed virtual file systems that represent abstract interfaces, and second, build the user environment by importing desired resources from the network. We have built the Plan B OS using this approach, its remote file system import mechanism allows the system to adapt to changes in the environment and permits users to customize the environment according to their needs. The system has been in use for over a year both to carry out our daily work and to implement the smart space that we built for our offices, which has machines with Plan B, Plan 9, Windows, Symbian, and Linux.

1. Introduction

It is hard to design, build, and deploy applications for pervasive computing, as pointed out by [12]. We built an environment [1-6, 26] that has multiple machines per user and many small devices attached to the network, to support ubiquitous computing [24]. Our main objective is to build a system *easy to program with* that could *interoperate well* with others, to make it easier to build smart spaces and to develop applications for them. The system includes services like location and context handling, motion sensors, voice interfaces, and X10 appliances. But instead of integrating multiple different systems with middleware, usually called a *meta* operating system [20], we tried to build an actual operating system that could cope with these environments.

Our system, Plan B, exports all resources to the network using abstract interfaces that are mapped to (distributed) file system operations [4]. All resources are seen as files, following the ideas in Plan 9 [18]. This includes important system services like application execution, user interfaces, context handling, event delivery, audio and voice facilities, and (physical) environment automation.

The aim of our system is to provide the mechanisms needed to build pervasive computing

† This work supported in part by spanish MCyT TIN-2004-07474-C02-02 and by Comunidad de Madrid project S-0505-TIC-0285.
‡ This author supported in part by spanish FPI grant BES-2003-2942.

environments, and to make it easy for the users to use and to develop new programs to implement and deploy the policies and tools chosen by the users. To achieve this, we built a system that both tries to bring back a single system image for each program (although such image differs from program to program, because each one can customize its environment).

Plan B is the first system that we used that made us think that all the machines each of us owns and all appliances present in the environment work indeed as a single environment. When programming or using the environment from any machine, we usually forget about which machines actually own the devices we are using, which is convenient to program applications for pervasive computing [25]. Our file system import mechanism permits the system to stay working and adapt when a machine is disconnected, a resource becomes unavailable, or a preferred resource is discovered. New machines brought online are quickly seen as integrated in the environment, because they start using the services already started in the space (e.g. voice notifications, audio players, and other tools). The ability to move user interfaces around and to start using different devices further helps to erase the barrier around each particular machine [6].

The system is very easy to program and customize, by implementing small shell scripts or new programs. Programmability includes services like context, location, power switches, motion detectors, and the like. For us, there is no difference between the development environment and the smart space around us.

We do not use middleware, but our system interoperates well. Most machines today are capable of providing/using remote files and we provide services as files. Therefore, most machines can provide/use Plan B services. Although we prefer to use Plan B terminals, we have Linux systems that use services from Plan B, and most of the department uses the files for our context service exported by the web. In the same way, the Plan B speech synthesis and recognition volumes actually rely on Windows and Linux tools.

2. Abstract Networked Resources as File Systems

2.1. Abstract Resources

Our approach is to provide an *abstract* interface for resources, amenable to users and applications, through (virtual) file trees. For example, unlike other systems that use files for interfaces, e.g. Plan 9 [18], we do not provide pixmaps or other low-level artifacts to support GUIs. We support UIs by providing a file tree where *each* UI element (i.e., widget) is represented by a different file.

Note that these files are not real files on disk, they are provided by resource servers, like done in Plan 9 or UNIX's `/proc` [15]. Note also that we do not propose to intercept file system calls. We propose to take the Plan 9 approach to the limit and provide (all) services by implementing tiny file servers. Therefore, these servers receive the RPCs for their FS calls, with no call interception required.

Resource semantics are left out of the system, which handles files as raw data. However, most of the information necessary to understand a resource is modeled by the file tree representing such resource. In our example, if a UI element (e.g., a row) contains other UI panels (e.g., buttons), the former element is represented as a directory that contains the files for the latter ones.

Besides representing the structure of the resource, the file tree also provides abstract operations on it. All the implementation for such operations is provided by the file tree server, i.e. by the resource provider, and becomes shared in a natural way. What can be done with a file (or a set of files) that represents a resource depends on the resource considered. If a UI element is a image, we can copy images (in Plan 9 format) to its data file to update the image on the screen.

Certain operations, hard to express by means of a file operation, are expressed by writing them in textual form into a file provided by the resource. In this case, such file *appears* to contain the set of operations that are in effect at the time. For example, a vector-graphics widget is represented by a file that appears to contain strings like "`arrow from 0,0 to 100,150 red`", and "`circle at 50,50 rad 15`". Reading the file reports the drawing in the widget, and suffices to permit users to grasp the semantics of the file in a natural way. The format used to provide information (when reading a file) is always that used to accept requests (when writing a file).

In many cases, file names are chosen by the software providing the service (e.g., files names for X10 appliances are similar to `pwr:136light` to provide a clue about what the file represents). Each service provider is free to impose its own conventions regarding file naming, because that is part of the abstraction.

In some cases, the resource provider only imposes a few restrictions (e.g., the UI service requires each widget to be represented by a directory whose name includes the type of widget). In a few cases, no restriction is imposed on file names (e.g., the context service allows any file name, each file represents a piece of context named by the user).

We apply the guidelines above to *all* system resources: The command execution service provides a file where command lines can be written; the voice interface provides a single `output` file where text can be written for speech processing; the audio interface provides a file where files in MP3 format can be copied to be reproduced; the mouse interface provides a file that reports mouse events when read; an X10 appliance for a light switch appears to be a file that may contain "`on`" and "`dim 50%`" (e.g., it would accept a write of "`off`" to switch off the light); context known for user `nemo` appears to be a directory "`/who/nemo`" with files like `status`, `location`, etc.; and the list goes on to include all other system resources and services.

2.2. Dynamic Environments

Exporting all services through an abstract interface is not enough for a pervasive environment, which is highly dynamic. A mechanism to discover the relevant file systems for resources of interest, and to import them into the name space of the client machine, is also needed. Our approach in this respect has been to implement a resource discovery protocol and to provide a way for the user to select which resources to use in the system we built.

The system helps in selecting which resources to use, but the applications are still responsible on how to handle failures and failover when a resource is lost in the middle of an operation being made by the application. Therefore, it should be noted that we are not providing transparent fail-over or related mechanisms. We only adapt the name spaces without placing that burden into the application. Section 4.3 further discusses this issue.

There is a compromise between introducing middleware and changing the system. We changed the system because we wanted to demonstrate that applications could be kept simple if the system adjusts the application name spaces without placing the burden in the application. However, adding a FS discovery protocol to any other system is not very difficult. Indeed, Windows and other systems already include their own ones [16, 20].

3. Why Use Files?

It is interesting to look at the proliferation of XML based interfaces. Many approaches using middleware rely on them for interoperability. In our opinion, one of the reasons for doing that is to bring back a universal interface that could be understood by most programs (e.g., the FS interface). We restrict ourselves to the basic operations found in all file system (open, close, read, and write) because they are understood no matter the particular implementation used by the system considered. We replace what would be represented by an XML tree in other systems with a file tree (the information that others provide using XML attributes and tags is provided in our approach by means of file names and text files to hold the attributes).

A properly designed file interface is easy to understand both by people and programs. Also, it does not require the introduction of a new technology or yet more software to permit both users and programs to use it. For example, consider the X10 light switch interface mentioned in the previous section. A user can browse (and search!) a set of files to locate the file representing a light switch using the Windows explorer, because X10 appliances appear to be files; they just happen to be implemented by a remote X10 driver software. Furthermore, the user can open the file with the windows Notepad, replace `on` with `off`, and write the file back to switch off the light. When using UNIX, `ls` and `echo` can be used instead.

Approaches using middleware-provided abstractions, e.g., Speakeasy [8] or the IWS [14], would require installing or downloading software in the client machine just to browse or to operate the device. Speakeasy in particular strives for spontaneous interoperability. It relies on code shipping to provide the code necessary to speak the protocol used by the device considered and also to provide an interface for it. But we argue that the client machine might be able to interoperate with the device using its own software, because the FS interface is already available and most (when not all) systems support remote file access.

The only requirement is that the particular interface considered, must be of a high level of abstraction (despite being modeled through a file interface, which might be considered of a low-level of abstraction). That is, the file tree used for the device, file contents, and data and control operations done through files must be abstract enough to be meaningful to users and programs when used from any other system.

Another argument supporting the use of FS interfaces is that it can make all the information in the system available for inspection. Simple programs like `ls` and `cat` suffice to inspect the system without using a complex (e.g., reflective) architecture.

A weakness of file interfaces is event delivery. A poorly designed FS interface for a resource may force the user to poll the file tree to discover events of interest. Other approaches make event handling straightforward, e.g. IWS [14], but the price paid is losing other benefits gained by using files as the ubiquitous resource interface.

A way to side-step the problem is to implement a event delivery service as a FS. Applications may create files on it to create event channels, and the files may be read to receive events and written to post events. We use this approach for services that are naturally modeled as events, e.g., mail notifications, location changes, and user requests. This is similar to what systems like [14] do to interconnect their services and resources. Another way to handle the problem is to provide a file within the device that reports events to any client reading it. We use this approach only where it seems obvious that the events file is actually a stream of events, e.g., for the mouse, keyboard, and similar devices.

4. The Organization of Plan B

To experiment with our approach, we designed and built the Plan B OS [1-6, 26]. Figure 1 depicts the model for the system. Once booted, a machine exports all its resources to the network. Each resource is exported as a tiny file system. The Plan B file systems have an associated name and a set of attributes. We refer to each one of these tiny file systems as a *resource volume*, and to the set of attributes as its *constraints*. All the machines are considered peers in that they run the same system software and their purpose is the same: To export volumes.

This does *not* mean that all machines in the environment must run Plan B. Any machine that can mount Plan B's files can use services from Plan B volumes. In the same way, any other system may run one or more processes to export some resources as files. Our approach is readily applicable to any other system. Nevertheless, we built and use Plan B because of its resource import mechanism, which is best provided by the system to make things easier for users and applications.

Several protocols are used to export the resources. The Plan 9 network file system protocol, 9P [27], permits machines to export and remotely use files. A suite of authentication protocols permit machines that implement volumes to authenticate users, and viceversa. The Plan B discovery protocol announces volumes to the network and allows Plan B nodes to discover new (and gone) resources. Some of the machines gateway to CIFS and NFS to permit other systems to use Plan B volumes.

The environment seen by an application is the set of files imported from the network into its name space. In plan B, such files include all resources used by the application, including the widgets in its user interface and the devices used. This is the Plan 9 approach [17] taken to the limit. The environment seen by a user is that seen by all the applications that belong to him/her.

A general purpose event delivery volume provides ports that can be used as event channels. Each message written to a port file is delivered to all the readers of the file. Besides notifying events of interest, e.g. mail arrival, this service is used to request visualization of urls, reproduction of songs, edition of files, and execution of programs.

4.1. From Plan 9 to Plan B

Most of the kernel is taken from Plan 9 as it is (box in figure 1), e.g., process and memory management are exactly the same. To convert Plan 9 into Plan B three things were done: First, we changed the heart of the system (the name space implementation) to use volumes and constraints; second, we redesigned system services to provide them through *abstract* FS interfaces that could be operated from anywhere in the network; third, we added a discovery protocol to the resulting system.

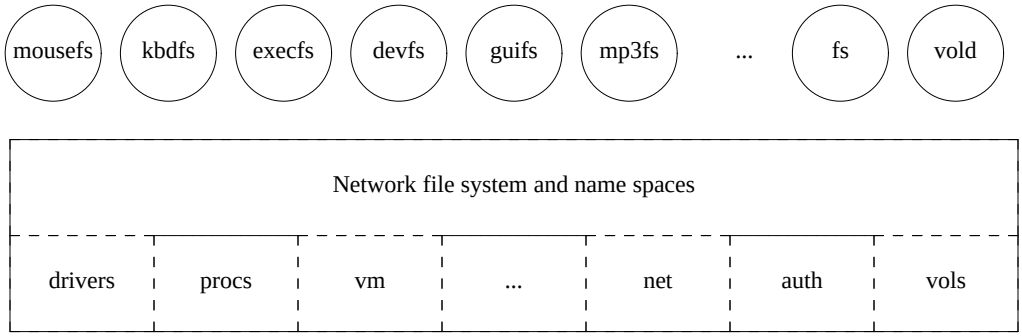

Figure 1: *Architecture of the system software at a Plan B machine.*

The Plan B *volume device*, `vols`, keeps the kernel aware of the set of known volumes. A user process, `vold`, implements a volume discovery protocol similar to others found in the literature (e.g., that in UPnP [23]). All volumes are implemented by servers that are user processes. Each server registers with `vold` to announce its service. Servers are also responsible for authenticating their users.

The network file system multiplexor implements the name spaces used by the processes and permits using 9P to reach the files and volumes bound to names in the space. File names are hierarchical paths similar to those in Plan 9 or UNIX. Processes use well known file operations, i.e. `open`, `close`, `read`, and `write`, to operate on the files that conform their environment. To implement these operations, the file system multiplexor maps them to volume operations and 9P RPCs.

4.2. Resource volumes

Volume names are strings that identify the resource exported in a global name space. The names for volumes are used to advertise them and to identify the volumes that are to be bound on import requests. Volume constraints are sets of attribute/value pairs that identify properties of interest[1]. The constraints are used to advertise the properties of the volumes along with its names, and also to request desired properties while importing resources. The system handles constraints syntactically, semantics are left out. Therefore, users and servers can choose which sets of constraints to use, and what do they mean (both parties must agree on the meaning of each constraint, though).

Constraints are usually defined by the server exporting the resource, but can be completed by machines in the path to the client. An interesting example is a constraint added by the client machine: `C`, which stands for *connection*. If `vold` needs more than 100ms to mount a volume (including the time to authenticate), a constraint `!Cbad` (i.e., bad connection) is added for the volume. This makes the system aware of which volumes are reached through connections with a very bad latency or a very poor bandwidth. Users can instruct the system to prefer volumes reached through a good connection for cases that matter, e.g. for system binaries. The next section shows how to do this.

4.3. Selecting Resources and Adapting

The primary means to ask for resources in Plan B is the mount system call, used both to import resources and to tailor adaptation. It instructs the system to search for volumes matching the given name and constraints, and import them at a given path. Once a volume has been imported, it remains in use until unmounted or becoming unreachable. Should this happen, the system would search again for an alternate volume that could be used instead.

Different resources can be mounted at the same mount point, in order of increasing preference, to tailor adaptation. For example,

[1] The syntax used for constraints in this paper is that used in the system when the paper was written: An initial letter identifies an attribute, and following characters denote its value. We have later changed this syntax to be `attr=value` (for usability).

```
mount -V /devs/audio!L136 /my/audio
mount -bV /devs/audio!L136!Unemo /my/audio
```

create a union of two volume mount entries. The first request imports any audio device at location `136`, and makes it available at the path `/my/audio`. The second one imports, *before* the previous one, any audio volume that besides the desired location is owned by `nemo`. This idiom permits the user to specify that the preferred volume is that identified by the second call. It also specifies that when the preferred one is not available, any one identified by the first call suffices as well.

This means that devices might change while an application runs, but only if the name space used permits that. Those (users or applications) that do not tolerate switching of resources can always explicitly import the resource of interest. Applications can learn of changes by keeping open file descriptors into the files used, because that descriptors would report I/O errors. As an additional aid, several files in the system report the set of volumes in use and the name space in effect. In any case, the system ensures that the name space reflects those resources available and cleanly removes from it the file trees for gone resources. If an I/O error happens while reading a directory or performing any other call, the system considers the involved (remote) file tree as faulty and ceases using it until it could be remounted.

What happens with server-side state depends on the service. For example, UI volumes connect to the application to report creation and replication of widgets (widgets are garbage collected when their application is not reachable), X10 appliances simply retain their state, audio players cease playing if the client is gone (the application would reopen the audio output file, which might be on a different volume), etc.

5. A Programmable Smart Space

The framework for context handling is simply a set of volumes containing files to describe context for users, places, and things. There are three volumes for maintaining context information: `/who`, `/where`, and `/what`. Each one is a file tree that keeps different context within different files. Location is handled in the same way. Also, most volumes include a constraint that specifies their current location, to permit users to import volumes depending on their location.

Presenting all resources as files that can be dynamically imported to match the user specified names and constraints is very powerful and permits programming the system in a simple way, as we show next. For example, the `tell` program is a simple shell script that accepts a user name and a text message. It imports any of the speech volumes sharing the location with the given user name, and speaks the message there. The involved commands are shown here.

```
# $user and $message given as arguments.
location=‘{cat /who/$user/where} # set $location to the user location
# import an speech volume from that location
mount -V /devs/voice!L$location /devs/voice
# deliver the message
echo ‘{who am i} to $user: $message >/devs/voice/output
```

Note how the different aspects of the system are combined together in a very effective way. The file interfaces are easy to use from application’s code and shell scripts. In the code we could see how location information (e.g., the piece of user’s context kept at `/who/$user/where`) is handled in a uniform way, and can be obtained by reading a file. The file import mechanism permits the selection of appropriate resources based on required constraints (e.g., location). Note also how new system services introduced for pervasive computing (e.g., voice and location) are handled as any other system service.

Recently, an active RFID system (using hardware from Hexamite) has been installed and a new tool has been made available to let users update their location according to their badges. No change had to be made to system (apart from implementing the corresponding file server for the new location volume).

User interfaces are provided by UI volumes, each volume handles a given screen and provides graphical widgets on it, as seen in figure 2. Each screen is handled through a file tree that represents the tree of widgets in the screen. The state of each widget is self-contained, and includes the network address of the application responsible for it. Therefore, the files for a widget can be copied to a different place (perhaps into a different device) to move or replicate all or part of a UI. Figure 2 shows replicated controls for the player program.

What is important for us now is not how the UI service works (you can refer to [6] for that), but how easy is for us to program with UI elements once they have been abstracted and exported through files. For example, this is the command used internally when the mouse is used to copy and paste a tree of widgets (a complete UI, part of an UI, or just a single widget):

```
# $from is the path for the copied UI tree
# $to is the path for the target UI location chosen by the user
cd $from
tar c *  | { cd $to ; echo hold >ctl ; tar x  ; echo nohold >ctl}
```

It can be seen how `tar` archives all the files for the copied UI, and how it is also used to recreate the UI at the target. What is more important, we could use `tar` in the same way to archive and recreate the state of X10 power switches for a room, or most other resources. In the example, the two `echo` commands are used to ask the UI to hold screen updates while extracting the UI, so that each individual widget created does not imply a screen update. Note how a hard to express operation, i.e. *hold*, can be easily expressed by means of control files.

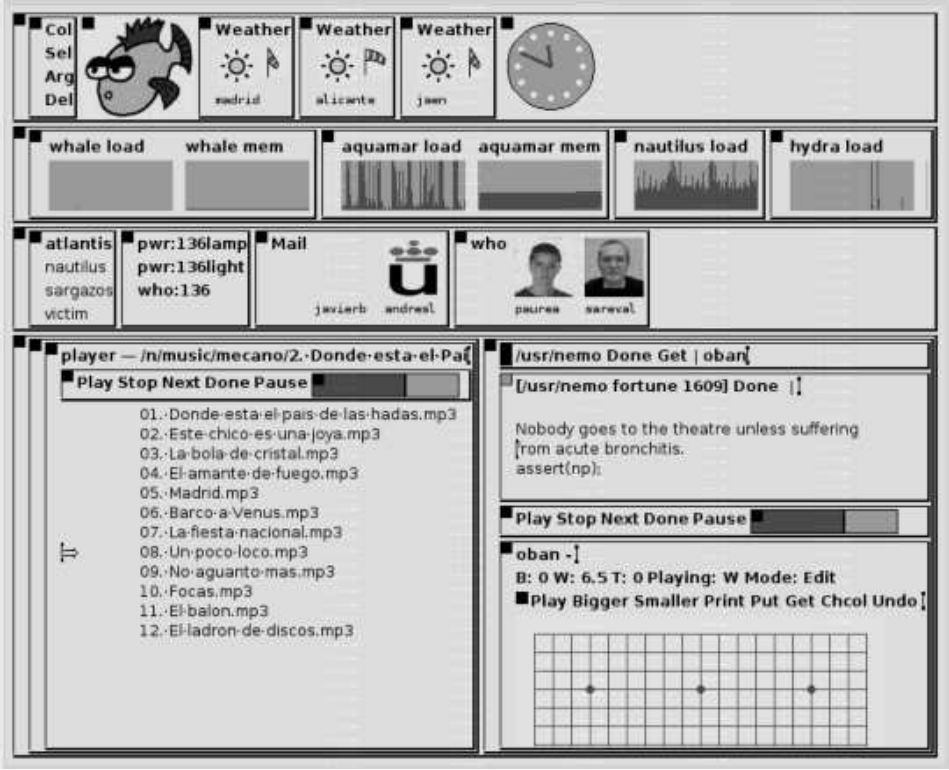

Figure 2: *A typical Plan B screen, serviced by the* `omero` *UI volume.*

As further examples, we can write very simple scripts to do things like lowering the volume level of a room if there are visits (reading the context file `visits` for the room and updating the `volume` file for any audio within that room):

```
# remember old volume level
old=‘{cat /devs/audio/volume}
location=‘{cat /what/$sysname/where}
if (grep yes /what/$location/visits >/dev/null){
        # lower it a 10%
        new=‘{expr $old - 10}
        echo $new >/devs/audio/volume
}
```

Determining if there are visits can be done also via a simple script, by looking at X10 motion detectors for the visitors area in a room:

```
location=‘{cat /what/$sysname/where}
if (grep yes /devs/x10/$location:* >/dev/null)
        echo yes >/what/$location/visits
if not
        echo no >/what/$location/visits
```

The scripts just shown can poll the file system for changes. However, in many cases, it is more convenient to use events. This script accepts events for image files to be shown at a big screen in the room:

```
touch /devs/ports/poster
while (;){
        cat /devs/ports/poster | page -w
}
```

And such events can be sent easily as well:

```
# mount event ports sharing our location
mount -V /devs/ports!L$location /devs/ports
# and post...
echo $file > /devs/ports/poster
```

As a further example, we can also pause any player when no user is in the room. To do so, the script searches user interface files in the room for a file representing a button (its name would start with `button:` as dictated by our implementation) with the name `Pause` and then it writes a control request to the widget to press the button.

This is very important for environment automation. We have found that the appropriate way to make the environment *intelligent* is not implementing a big application that automates most things, but implementing many small tools instead. Each tool inspects the environment, makes a choice and updates the environment. To construct the tools, it is very convenient to be able to use general purpose programs. Otherwise we would have to write multiple specific purpose programs, perhaps using a programming language designed just to do that, like in [19].

6. Security and protection

Authentication and protection are handled as in any traditional distributed file system. We use the Plan 9 authentication service [18]. which does not require an unique administration domain, and is known to scale well. The ownership and access control lists provided by the file systems in the Plan B volumes are the mechanism we use. For example, this command allows any user to switch on/off the lights at room 136, yet prohibits reading the light status to casual users:

```
chmod a+w o-r /devs/x10/pwr:136light
```

Users who want to grant access to users (i.e. badges) near a device, can do so by reading the context files for location and updating the permissions for the device files accordingly.

7. Lack of type checking

Control operations are codified as text, as is status information. Therefore, there is **no** type check for control operations. Nevertheless, years of experience with Plan 9 and Plan B have shown that this is not a problem in practice. When a erroneous control operation is issued to a control file, the file system responds with a `unknown control request` diagnostic, and the user corrects the mistake.

8. Experience and Lessons Learned

We have been using our smart space for daily work as our computing system for more than a year and a half. There are several demonstrations at `http://lsub.org`. The main lesson we learned from experience is that 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, can be more powerful than big software frameworks. They are simple to implement, easy to use, and need no further system services. Providing the services at the system level made all the existing applications able to exploit the new environment.

Programming applications for the smart space is done exactly like programming applications for a single PC. The code may perhaps write voice messages, turn on the lights, or check out the availability of its user; but that is the only difference. When the application is launched inside the space, its name space determines which resources are used and when to use others. The user has complete control over the distribution of the interface and other resources, and can control the application remotely, either using the file systems or a mouse and keyboard in the network.

The hard work is shifted from programming applications to thinking on an appropriate (FS based) interface for each new device and service. This has always been a problem no matter the paradigm used. However, in Plan B, the consequences of a bad interface are more serious, because of the lack of types in the system other than those implied by the hierarchy of files used.

Our system usage shows that it scales well at least for serving a department. Client machines discover and mount only those resources they need. Therefore, file systems do not end up with too many clients and a high load, and client machines do not need to mount a myriad of different servers to operate. But we do not know if the system would scale beyond a typical department or group of users.

There are two things that are not solved in a fully satisfactory way. Both apply only when using the system from non-Plan B machines (e.g., Windows). First, other systems must gateway through Plan B nodes to benefit from the volume discovery service. Second, using files may be inconvenient for the end-user to operate some resources. In this case, simple programs must be developed to can a UI for the user.

9. Other approaches

Related work is too abundant to be described here. We mention here only the most representative related work, and focus just on the main differences.

Plan 9 [18] is a distributed system that is built by exporting all resources as files and allowing those files to be accessed through the network. Plan B borrows many of the Plan 9 ideas. There are some important differences though. Plan 9 does not adapt to changes in the environment. Many times, changes require user intervention, and some times they require rebooting the machine. Plan B uses dynamic volumes to adapt to the environment. Furthermore, important system services have been redesigned with portability and adaptation in mind. Besides, Plan 9 lacks a mechanism similar to the constraints we use to select which resources to use.

The Semantic File system [10] and Globe's name service [16] are able to select resources that present a set of properties by means of attribute-based queries. However, they use a very different approach for exporting system services, usually through typed interfaces or distributed objects.

Globe [22] and other systems, like Speakeasy [8], Ninja [11], Gaia [20, 21], IWS [14] and One.World [13] rely heavily on middleware as the means to implement and distribute their services. A big difference between middleware based systems and the approach shown in this paper 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 in middleware based approaches, ours

permits a native Windows or Symbian applications to access the new services just by using the file system interface. For example, Gaia had to introduce a scripting tool [20, 21] to simplify the use of their system, we can simply use the OS shell. Gaia's Context FS [7], which provides a file server interface for context based queries, is closer in spirit to our approach. However, we use files for *all* system services and do not require middleware.

There is plenty of work about how to use XML and related markup languages to exchange data and support interoperation. See for example [9, 23]. 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.

Overall, the main difference between Plan B and other systems is that by mapping abstract interfaces to files, and adapting to file tree availability, we provide a system that is easy to program, offers a general purpose computing environment, works for legacy applications, and supports smart spaces for programmers, without using middleware.

10. Future work

Most of the pending work refers to new programs needed and to experimentation with new policies and heuristics for handling context and making the environment smart. We are satisfied with the organization of the system and the interfaces developed for its services, and do not expect many changes in this respect.

References

1. K. L. Algara, F. J. Ballesteros, E. S. Salvador and G. G. Múzquiz, UbiTerm: A hand-held control-center for user's activity mobility, *IEEE International Conference on Pervasive Services*, Juy, 2005.
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. F. J. Ballesteros, E. M. Castro, G. G. Muzquiz, K. L. Algara and P. H. Quiros, A New Network Abstraction for Mobile and Ubiquitous Computing Environments in the Plan B Operating System, *Proceedings of the IEEE WMCSA*, 2004.
4. F. J. Ballesteros, E. S. Salvador, K. L. Algara and G. Guardiola, Plan B: An Operating System for Ubiquitous Computing Environments, *Submitted for publication. Also at http://lsub.org.*, 2005.
5. F. J. Ballesteros, G. G. Muzquiz, E. Soriano and K. L. Algara, Traditional Systems can Work Well for Pervasive Applications. A Case Study: Plan 9 from Bell Labs Becomes Ubiquitous., *Percom*, 2005.
6. F. J. Ballesteros, G. Guardiola, K. L. Algara and E. S. Salvador, Omero: Ubiquitous User Interfaces in the Plan B Operating System, *Submitted for publication. Also at http://lsub.org.*, 2005.
7. C. K. H. R. H. Campbell, A Context File System for Ubiquitous Computing Environments, *Technical Report No. UIUCDCS-R-2002-2285 UILU-ENG-2002-1729*, July.
8. W. K. Edwards, M. W. Newman, J. Sedivy, T. Smith and S. Izadi, Challenge: Recombinant Computing and the Speakeasy Approach, *8th ACM Mobicom*, 2002.
9. 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.
10. D. K. Gifford, P. Jouvelot, M. A. Sheldon and J. W. O. Jr, Semantic file systems, *Proceedings of 13th ACM Symposium on Operating Systems Principles*, 1991, 16-25.
11. 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), .
12. R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad,

G. Borriello, S. Gribble and D. Wetherall, System Directions for Pervasive Computing, *Proc. IEEE HotOS-VIII*, May 2001.

13. 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), .
14. B. Johanson, A. Fox and T. Winograd, The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms, *IEEE Pervasive Computing Magazine*, April 2002.
15. T. J. Killian, Processes as Files, *Proceedings of the Summer 1984 USENIX Conference*, 1984, 203-207..
16. I. Kuz, M. Steen and H. J. Sips, The Globe Infrastructure Directory Service, *Computer Communications 25*, 9 (June 2002), .
17. 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.), .
18. R. Pike, D. Presotto, K. Thompson and H. Trickey, Plan 9 from Bell Labs, *EUUG Newsletter 10*, 3 (Autumn 1990), 2-11.
19. A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell and M. D. Mickunas, Olympus: A High-Level Programming Model for Pervasive Computing, *Proceedings of 3rd IEEE Intl. Conf. on Pervasive Computing and Communications*, 2005, 7-16.
20. 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.
21. M. Roman, An Application Framework for Active Spaces, in *PhD thesis*, University of Illinois at Urbana-Champaign, 2003.
22. M. Steen, P. Homburg and A. S. Tanenbaum, Globe: A Wide-Area Distributed System., *IEEE Concurrency*, Jan-Mar 1999.
23. uPnP-Forum., Understanding uPnP., *www.upnp.org*, 2004.
24. M. Weiser, The computer for the 21st century, *Scientific American*, September 1991, 94-104.
25. M. Weiser and J. Brown, Designing Calm Technology, *PowerGrid, Vol 1*, 1996.
26. Plan B User's Manual. Third edition., *Laboratorio de Systemas, URJC. GSYC-Tech. Rep.-2005-04. Also at http://planb.lsub.org/sys/man.*, 2005.
27. Plan 9 Programmer's Manual, *AT&T Bell Laboratories. Murray Hill, NJ.*, 1995.