Plan B: An Operating System for Ubiquitous Computing Environments*

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

Abstract

The conventional approach for building pervasive environments relies on middleware to integrate different systems. Instead, we have built a system that can deal with these environments by exporting system resources through distributed virtual file systems. This requires no middleware, simplifies interoperation, and permits applying general purpose tools to any system resource. A constraint-based file system import mechanism allows the system to adapt to changes in the environment and permits users to customize the environment and tailor adaptation according to their needs. The system has been in use for over a year to carry out our daily work and is underlying the smart space that we built for our department.

1. Introduction

The problem we address is how to provide a convenient operating system for a ubiquitous computing environment [?], which includes multiple machines per user, small devices attached to the network, and new services like location and context handling that are necessary for its applications. The environment is also highly dynamic, and includes mobile users and devices. Therefore, the system must enable adaptation.

Most other systems use middleware to provide new abstractions designed for this kind of environment. We do not. Our approach, Plan B, is to split the system to export all its resources to the network using abstract interfaces that are mapped to file system operations. Demonstrations of the resulting system (and our smart space) can be found at `http://lsub.org/ls/planb.html`.

*This work supported in part by spanish MCyT TIN-2004-07474-C02-02. E. Soriano also supported by FPI grant BES-2003-2942.

All resources are seen as files, following the ideas in Plan 9 [?] (our implementation derives from Plan 9, indeed). A network file system protocol, 9P [?], is used to interconnect the system. The computing environment is built for each user by importing the required resources from the network. Any machine that knows how to use/export remote files can interoperate and be integrated in our environment, and even mobile phones can do so using OBEX on bluetooth or Infrared links. On the other hand, systems that use middleware need to deploy additional services, e.g. Gaia's microserver [?], into devices to interoperate well.

We do not have to introduce specific languages to simplify the programming of the system, e.g. Olympus [?] for Gaia [?], we can use the shell, C, and any other language that knows how to use files.

The import mechanism is built in a way that permits users to specify what they want yet lets the system choose what resources to use depending on which ones are available to satisfy the requests of the users. This makes adaptation to changes easier: If a resource becomes unavailable, another one, which must still comply with what the user requested, may be used instead.

Resource interfaces have been designed from the ground up to be of a high-level of abstraction, to tolerate a high heterogeneity of resources. Their high-level of abstraction is also an enormous aid for interoperability. For example, we use MP3 as the format for audio output because players are ubiquitous.

The main contribution of this paper is the description of a way to organize the operating system that avoids the need for middleware in platforms for pervasive computing and smart spaces. The consequence is that we can use general purpose and existing system tools, and we can keep the system simple.

More precisely, in this paper we describe the Plan B resource import mechanism, and the design of several important system services including application execution, user

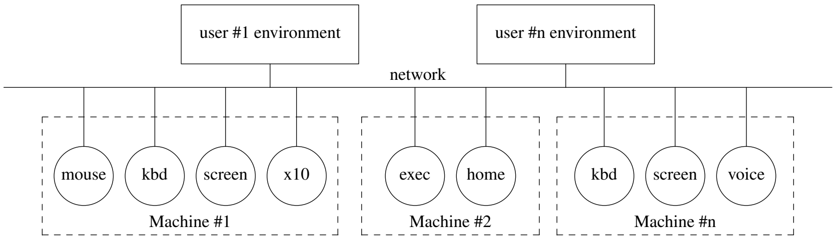

Figure 1. A Plan B computing environment is built out of individual networked resources.

interfaces, context handling, event delivery, audio and voice facilities, and (physical) environment automation.

2. System architecture

The overall organization of the system is depicted in figure 1. Initially, each machine (i.e., all its devices and resources) is owned by the user who boots it. Once booted, a machine exports all its resources to the network. Each resource is exported as a tiny file system that has 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.*

Most of the files in resource volumes are not real files, but virtual ones. This is similar to the `/proc` [**?**] in modern UNIX systems and to most files in Plan 9 [**?**]. For example, the `kbd` (keyboard) volume appears to be a single directory with two files on it: `cons`, and `kbdctl`. Reading from `cons` retrieves runes (characters, usually) from the corresponding keyboard. Writing to `kbdctl` permits issuing control operations to it, e.g., to redirect the keyboard for use with another GUI.

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. For example, all audio output volumes have the name `/devs/audio`. Our convention is to name each volume after the path where it uses to be mounted, but any other convention can be used as long as it is consistent.

Along with the name, each volume has constraints. The constraint set for each particular volume contains a set of attribute/value pairs that identifies properties of interest. The constraints are used both to advertise the properties of the volumes along with its names, and to request desired properties while importing resources. Unlike other systems, e.g., INS [**?**], we do this for *all* resources.

Constraints are represented by a string that contains attribute/value pairs separated by a `!` character. For example, `!Hnautilus!Unemo!Tmp3!Cbad`. Each attribute/value pair is represented by an initial upper-case rune that identifies the attribute and a lower-case string that represents its value[1]. In the example, `H` is the attribute that names the machine providing the service, and `nautilus` is its value. We call `!Hnautilus` a constraint.

Once an application is started, it is bound to the machine where it was created. Usually, the machine where the application is created is the one where the request was issued from. Most machines have plenty of processing capacity to permit this. For the cases when a machine has limited processing capacity, the requests to start new applications are delivered to another machine owned by the same user. Process management is therefore exactly like that of a centralized system. There is no process migration nor code shipping (other than paging a binary through the network). As a result, unlike other systems [**?**], we do not have to face the issues raised by such techniques, most notably security problems.

Each process has a name space that binds names to resources, i.e., to files and volumes [**?**]. This feature permits the user to adjust the environment of the process. Some of the binds refer to a concrete file at a particular file tree, and they work like traditional network-mounted file systems. Other binds refer to volumes and request the system to bind to a file name whatever volume (or set of volumes) satisfies a given volume name and constraints.

The environment seen by an application is the set of files imported into its name space. The environment seen by a user is that seen by all the applications that belong to him/her. This is similar to what happens in Plan 9 [**?**], if we forget about the different interfaces for system resources that we use and forget also about the environment changing underfeet (e.g., when a device is replaced with another, or when switching off the network in a laptop).

The system software running at a Plan B machine is shown in figure 2. The boxes represent kernel components and the circles represent user processes. Most of the kernel is that of a Plan 9 machine. In fact, the implementation of

[1] In the current version of the implementation, we use the syntax *name=value*, but the idea remains the same.

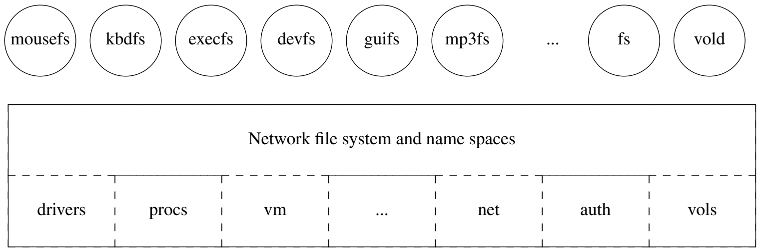

Figure 2. Architecture of the system software at a single node.

Plan B (3rd edition) is a Plan 9 descendant. Process and memory management is exactly the same. Drivers for I/O devices that are not exported verbatim to the network are kept within the kernel as well, including the network protocol stacks.

An important driver is `vols`, or the *volume device*, that keeps the kernel aware of the set of known volumes. The driver exports a single file `/dev/vols` that can be read to see what idea the kernel has of existing volumes. A write to the file can be used to ask the kernel to add or delete volumes. This task is usually performed by `vold`, a user process that implements a volume discovery protocol similar to any other found in the literature.

The central part of the kernel is the network file system multiplexor. It 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.

3. Importing volumes

The core of the system is the mechanism used to import volumes into the name space of the application. The elements involved are shown in figure 3.

- Each volume server exports its files and registers with a discovery service (e.g., the volume daemon) to export the name, constraints, and network address where it can be reached.
- The volume daemon (or any other user program) informs the kernel of volumes discovered and probes them for availability. When a volume becomes unreachable, the volume daemon informs the kernel as well.
- The volume driver keeps a table of known volumes. It is the interface between the kernel and the discovery service.
- The name space implementation translates path names to files. Some of the files correspond to files within volumes imported.
- The user programs rely on the `mount` system call both to mount file systems and to import volumes.

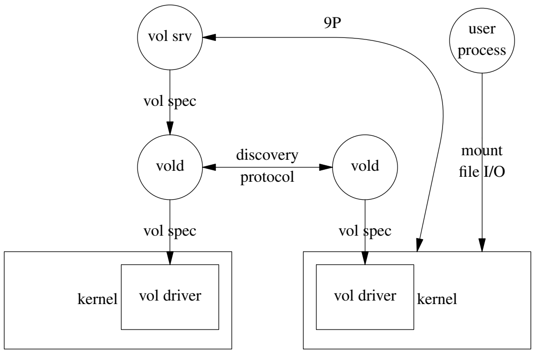

Figure 3. Elements involved in the volume import mechanism.

The mount system call can be used in three ways. The first one is similar to that used in Plan 9, and installs a new entry in the name space to map a name to the root of the file tree serviced at the other end of a given network connection. The second way to use the call is analogous, but imports volumes instead. For example,

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

adds a mount entry that binds the name `/my/audio` to any volume named `/devs/audio` whose constraints satisfy `!Unemo!L136` (the owner must be `nemo` and the location for the device must be `136`).

To satisfy this request, the kernel scans the volume table kept in the volume device for any matching volume. A volume matches if the name is the same and the constraints have at least the requested attributes and values. The first volume found is automatically mounted on behalf of the user. If, later, the kernel is informed that the device is gone (or suffers an I/O error trying to reach a file in the volume), the volume is unmounted and the volume table is searched again looking for another volume that also satisfies the name and constraints. Should such a volume be found, it is mounted without any intervention of the user. When no volume satisfies the mount request, an empty directory is mounted instead (while waiting for a matching volume to be available).

The last way to use `mount` is to request a union of volumes instead of importing a single one. For example,

```
mount -U /devs/ui!Unemo /my/uis
```

requests all known volumes with name `/devs/ui` (i.e., any user interface service), and a constraint matching `!Unemo` (i.e. owned by `nemo`), to be imported at the name `/my/uis`. After the call, the mount point appears to be the union of all root directories for the volumes mounted. Note that the set of volumes mounted may change due to volume availability. Initially, the kernel scans the volume table and mounts *all* matching volumes. Later, volumes known to be gone are unmounted automatically. Also, any new volumes found in the future that match the request will be added to the union without user intervention.

Processing for both mounting volumes and mounting volume unions happens on demand, as processes try to resolve names. A serial number attached to both the volume table and the name space(s) data structure is used to determine if the latter must be updated. System calls resolving names perform this check and any further updating of the name space.

Each one of these modes of using `mount` may be further qualified by using two other flags to request mounting *before* or *after* the previous contents of the mount point. Indeed, mounting something above/below creates a union. The new mount entry is added before or after the previous ones depending on the flags. This permits the user to override contents of a given directory depending on the volume availability. For example, these requests

```
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`. 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. Note that this works because the a mount that has no matching volumes works very much like an empty directory.

Each process, like in Plan 9, provides a textual representation of its name space that can be read from its corresponding `/proc/$pid/ns` file. This makes the user aware of the concrete resources (i.e. volumes) being used, which is necessary to avoid paranoia regarding volumes changing underfeet.

3.1. File descriptors and errors

When the kernel updates the name space for a process, file descriptors kept open by the application are untouched. File I/O is kept to going to the same file even if the volume containing the file is being replaced in the mount table with another one. If the old volume is really gone, file I/O will fail and an error is returned to the application. If what happens is that a preferred volume has been discovered, file I/O will continue in the old volume until the file descriptor is closed.

Two new system calls[2], `readf` and `writef`, receive file names instead of file descriptors. It is up to the applications to keep file descriptors open, or to use two new calls to get/put all the file contents. For example, audio players use `writef` to write (portions of) MP3 files to audio devices. This is reasonable because when an audio device is gone it is sensible to try with another one. Compilers, on the other hand, use `write`, and keep the object file open. If the volume containing the compiled files becomes unreachable, the user can see that the compilation failed (and why) and take an appropriate action.

4. System services services revisited

In this section we describe several of the system services. Those not mentioned here are designed along the same lines. For example, see [**?**] for a description of our prototype for a network programming service. Further information can be found at [**?**].

4.1. Application I/O and user I/O

The set of I/O devices used by applications is determined by the volumes mounted, which may change during the execution of the application. This provides a mechanism to redirect I/O to appropriate devices based on volume names and constraints.

Pointing devices and keyboards have respective volume servers to be exported to the network. Mouse volumes accept calls from the network from peer mouse volumes. The resulting connections are used to forward mouse events from one mouse volume to another. The user may control mouse forwarding by writing to the `mousectl` file supplied on each mouse volume. A click in the upper left corner of the screen reclaims the mouse and breaks any redirection

[2]Not real system calls, because they are implemented within the C library.

being made. The keyboard volume speaks a similar protocol, and can forward runes to a peer server. In this case, a escape key can be used to reclaim the keyboard.

As a result, in Plan B we can easily handle multiple screens either using a single keyboard/mouse or using multiple ones. All of the text/pointing input devices become now similar. They provide input to the environment. To which screen, it depends on the preferences of the user at any time.

4.2. Audio devices

Audio devices are managed by the in-kernel driver and are exported through volumes that present a higher level of abstraction. Currently we use MP3, speech recognition and synthesis volumes. This makes it easy to build new useful applications.

For example, the `tell` program 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 $msg given as arguments.
# set $location to the user location
location=`{cat /who/$user/where}
# import a speech vol. from there
mount -V /devs/voice!L$location /devs/voice
# deliver the message
echo `{who am i} to $user: $msg >/devs/voice/out
```

4.3. User interfaces

User interfaces are provided by UI volumes. The preferred implementation for the service is a program called `omero` [**?**], and we use this name to refer to an UI volume. Omero handles a given screen and provides graphical widgets on it. It takes its mouse and keyboard from other volumes. Figure 4 shows a typical omero screen (multiple such screens are usually available).

In omero, each widget is represented or exported by a directory that appears to contain a `ctl` and a `data` file. Rows and columns contain (sub)directories for the widgets they contain. This leads to a file tree that mimics the tree of widgets seen on the screen. To create a widget, a program creates a directory. Removing it, removes the widget from the screen. The name of the directory determines the type of the widget. For example, rows have names that start with `row:`, column names start with `col:`, and so on.

Omero permits the user to split and recombine portions of the user interface using commands or the mouse language to cut, copy, and paste. Because the state of each widget is self-describing, the files of a widget can be copied to a different place (perhaps into a different UI volume) to move or replicate all or part of a UI. Figure 4 shows replicated controls for the player program (one replica is within the player UI, the other is just above the Go board shown by oban). This is further described in [**?**].

For most widgets, the data file contains a textual representation of its contents. The control files contain the attributes of the widget (and permit updating them). Most editing is handled by omero itself, the events seen by the application are of a high level of abstraction. But for the graphic widgets, events mostly consist on requests to *look for* a string or to *execute* a string. This makes many applications unaware of the mouse and permits a low bandwidth connection between omero and the applications using it.

4.4. Sensors and actuators

Our approach for handling these devices is exemplified by the X10 volume [**?**]. It provides files to handle X10 movement sensors and power switches. Client machines can simply mount the X10 volume(s) to use it. Each file in the volume (e.g., `pwr:136light`) represents an appliance and appears to contain the string `on` if it is switched on, and the string `off` otherwise. The file names used by the volume are configured when X10 devices are installed, they correspond to appliances controlled by the X10 devices. Their ownership is set to correspond to the owner of the physical space (who could be a user group). Note that for more complex appliances, the format of the file may be more complex (e.g., multiple lines with `attribute:value` entries), or the interface may rely on using multiple files (e.g., like shown for Omero in the previous section).

4.5. Context handling

The framework for context handling is simply a set of volumes that contain files to describe context for users, places, and things. The overall organization is described in [**?**]. There are three types of volumes for maintaining context information: `/who`, `/where`, and `/what`. Each one is imported into a union of volumes that is bound to the file of the same name. Permanent context is kept in actual file systems, and volatile context uses to come from file systems kept in RAM. Unlike in other approaches [**?**], the constraints mechanism to import volumes is kept separate from the context framework and works on any system resource. Also, defining new context can be as simple as creating it with the editor or a shell script.

A `/who` volume contains a single directory with the user name, and several files inside that hold separate pieces of context. Most notably, the file `/who/user/where` reports the last known location for `user`, and the file `/who/user/status` reports the user status using a description similar to the one used in the Instant Messenger. The `who` application seen in figure 4 (i.e., the faces) shows the list of users in the system, like the UNIX program of the same name. The context volume permits the tool to show

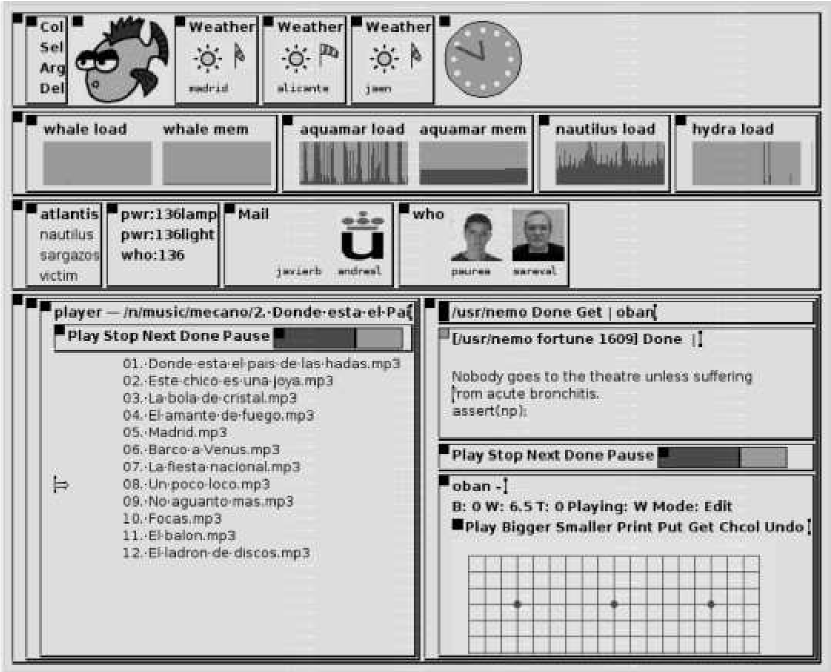

Figure 4. A typical Plan B screen, serviced by the `omero` UI volume.

users in the space, and not just in the system. Some of the users do not even use Plan B, but we collect context for them as well. Context for places and things is handled in a similar way.

Using context is fairly simple in Plan B: We read files. Listing `/what` gives an idea of the machines present in the environment. Using a program like `grep` permits a user to find who is busy and present in the environment, or which users are located where. We already saw how `tell` used `/who` to locate a user. But we did not show that the script can actually refuse to work if the target user is busy.

To extract and combine the context information various tools are used. Users start their preferred tools either manually or from their profiles. Our `fcompose` program [**?**] implements a very simple pattern language to generate strings depending on the contents of existing files. This suffices to generate context derived from existing information in most simple cases.

More elaborate context is usually extracted by means of heuristics that are implemented in tiny tools. For example, movement in the visitor's area of a room (reported by X10 volumes) uses to mean that the room has visits and the owner is busy (attending the visits). A public view of part of the context for our users, including unread physical mail, can be found at `http://lsub.org/who`.

An important piece of context for machines is which role do they have for their user. The roles are stored as strings that, among other things, are used by the user's profile and user commands to determine which applications should start where. Roles are determined by executing programs in Prolog and Gofer that use context for machines to determine the role for each one.

As an example of how we use roles, we use to start file viewers on the main screen (the largest one available). Editing uses to take place on the primary screen (the largest one that is known to be in the desktop), and other screens are considered auxiliary. The particular roles are important only for the user that owns the devices, which makes it easy to let different users assign different sets of roles to machines.

4.6. Location

Location is kept both in context volumes for the involved users and things, and also in constraints for volumes. Composing and generating location information is done in the same way used for other context. We keep the information in files. There are several tools to update location information and users choose which ones they use or write their own ones. The point is not which tool is used or how does it work. The important point is the system organization, i.e., how using volumes makes it easy to use and combine tools.

The X10 [**?**] service is being used to locate the owners of several rooms that are not shared. These users consider that when there is movement in their rooms, it has to be them. A simple shell script updates location by looking at files in the X10 volume. Other users consider that their location is that of their laptops, and a single script that pings for them updates their location files (in the `who` directories). Another script exists for users wearing RFIDs/badges, because using the mouse or the keyboard requires being physically close to the machine, and the badge location service reports the location for the user. This service uses hexamite hardware [**?**], and uses a file system as its interface.

The machine boot process asks for the location if there is no location configured. When a machine changes its location, the user must update it either manually or using a program started from the user's profile.

4.7. Events

A general purpose event delivery volume, `portfs`, 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. The scheme is similar to the Plan 9's plumbing service [**?**], but has been heavily modified for Plan B.

Most name spaces use to import to `/mnt/plumb` a union of port volumes that share the location with the machine where the involved process(es) runs. In this way, `/mnt/plumb` contains all the files used to deliver events to any of the programs sharing the physical space. It is customary to import the local port volume before importing remote ones. This gives priority to local ports over remote ones.

The API for the event system is provided by a separate file system, `netplumber`, that provides a single `send` file. This file system is not a volume, and runs for each machine involved. Its purpose is to process event messages written to it and then choose an appropriate port to deliver the message. The `send` file is local, but the port used may come from any user device.

The configuration file used to start the `netplumber` may instruct the service to start the appropriate application when no-one is listening at a given port. For example, the first time a song is requested to be reproduced, a player is started because the `song` port does not exist yet. The player creates the port and listens for messages coming from it. Any further request to play a song, from any terminal sharing the location with the player, will be delivered to the `song` port that is now being serviced by its program.

An interesting port is `exec`. It accepts messages sent from user interfaces to request the execution of programs. Most of the times, the port is serviced by the local `portfs`, which means that the program is started locally. However, when it is convenient, we don't service the port locally, which makes `/mnt/plumb/exec` to be resolved to a remote port instead. The result is that applications are started at a remote machine. On very slow modem connections, this is useful to start the user interface at the remote machine, and execute all the commands on a machine better connected to the rest of the system.

5. Common tools and environment automation

Our approach permits using general purpose tools to operate on a wide variety of things. For example, `tar` or `zip` can be used to store the state of X10 power switches, or omero user interfaces, and restore them later either at the same place or at a different one [**?**, **?**]. This works because the interfaces for the involved services have been designed very carefully to be self-describing. For example, should omero have included an `events` file for each widget, using tools like `tar`, `grep`, and the like would not be feasible: They would block reading the events file.

The simplicity for the interfaces permits doing things like using the Windows Notepad to switch off the lights of a different room, just by editing the X10 file for the corresponding power switch (writing the text `off` into it).

This is important for environment automation. We have found that the appropriate way to make the environment *intelligent* is not to implement a big application automating many things, but implementing many small tools instead. Each tool inspects the environment, makes a choice, and updates the environment. Some of the tools are one-shoot while others stay running to adjust the system every once in a while. To construct such tools, it is very convenient to be able to use general purpose programs.

We write and use very simple scripts to do things like lowering the volume level of a room if there are visits, or pausing any player and switching off the lights if no user is in the room.[3] Being all resources files, all the system is available for inspection and use, without requiring any middleware.

6. Security and protection

Using files solves much of the needs for security and protection. This differs from using middleware to export new services, which usually leads to new security issues.

Authentication is performed by the file servers involved and the kernels of the client machines. We use the Plan 9 authentication service [**?**]. This service relies on a central authentication server to secure connections between clients and servers. We are now developing a peer to peer security

[3] To pause any player, the script scans the user interface volumes for `Pause` buttons, and issues a control request to the widget to press the button. This is not a 100% accurate, because the interface may be controlling players outside the room, but it works most of the times.

architecture, intended to provide a decentralized authentication service centered around humans.

Regarding authorization and access control, the ownership and access control lists provided by the file systems are the mechanism we use. For example, to allow public reading of a user's status, he can run this command:

```
chmod o+r /who/nemo/status
```

Ownership is assigned by the users who start the services. For services shared among users ownership is assigned depending on who is using the resources.

7. Interoperability

For the most part, interoperability is granted because most machines today are capable of remotely using files. Because 9P is not a widely used protocol, some of the Plan B machines run gateways that export Plan B files through CIFS or NFS. This adds Windows and UNIX to the list of systems that can use our files. Authentication with these systems is performed in the same way used for sharing real (i.e., on disk) file systems. Space visitors may also use guest accounts. Using services provided by other systems from Plan B machines is feasible as well. For several services, volume file servers are available to be run on UNIX and export some of its devices to the Plan B network.

As of today, 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. Besides, our speech synthesis and recognition volumes rely both on Windows and Linux tools. The I/O device redirection facility, the event service, and shared files for user directories give the illusion of using a single system.

8. Problems and drawbacks

System conventions must be respected for the system to work. Anyone using existing services must adhere to their conventions. The one who introduces a new service is the one who defines such conventions. For example, the only place to obtain the location for a user is the `where` file in the `/who` volume, and all programs must respect this. If a new piece of context, say *user disabilities* , is needed, a new convention will be established to determine how to store it. But note that this happens also for any other system.

Type checking while using resources is mostly reduced to the hierarchy of files used for each one. Control operations use to be codified as text, as is status information. Therefore, there is no type check for control operations. 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 uses to complaint with a `unknown request` diagnostic, and the user corrects the mistake.

9. Experience and evaluation

There are several demonstrations at `http://lsub.org` that provide some qualitative evaluation. Our experience says that files are powerful. Specifically when they are used for devices and not for data on a disk. It becomes almost straightforward to program, debug, and use the system because all it requires is using files, usually by means of existing general purpose tools.

It does not make much sense to perform quantitative experiments to compare the performance of Plan B to other systems mentioned in the related work section, because most differences are qualitative and are not introduced for performance improvements. Nevertheless, we include some measures made to give a glance of the performance of the overall system and compare with Plan 9 running on the same platform (Comparing with other systems would measure differences in the application code more than changes in the system). Measures are the mean of 5 experiments performed on a 2.4Mhz Pentium Xeon PC with 512 Mbytes of main memory and a 100Mbps ethernet connection.

The time to compile the kernel source is 31.28s when using Plan B, and 30.36s when using Plan 9 on the same hardware. The source was approximately 100.000 lines of C code, including the drivers. This shows a 3% of slow down for Plan B with respect to Plan 9. Taking into account that we used the same binaries for the compilers, the difference is due to the changes introduced in the system.

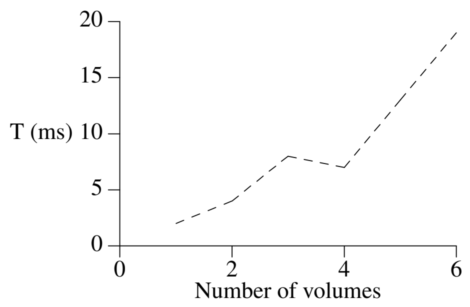

Figure 5. Time in ms to stat all files in the root of N unioned volumes.

The average system load is 15.2% on a Plan B system and 1% on a Plan 9 system, where 100% means that the CPU is always busy. This corresponds to a system running the window system, an editor, a mail reader, a load meter, and standard system processes, but idle otherwise. The increased load is the price we pay for having processes that monitor the environment, and for changing the interfaces for services.

Figure 5 shows the time in milliseconds to list the contents for a union of volumes (i.e. to read their root directories). It can be seen how adding more volumes to the union increases the time needed in a linear way. This is what could be expected, because reading a union of N volumes requires reading N root directories across the network. The time needed to search for a given file depends on the position of the file's volume in the union. All the volumes are searched in order until one is found. Measurements show exactly the same values for Plan 9 file mounts, which means that the volume mechanism did not add a significant performance penalty to the name resolution mechanism.

In the light of these experiments, we think that the 3% of slowdown seen while compiling the system kernel is due to additional mounts performed by the Plan B profile (with respect to the one used for Plan 9), and also to a higher system load while idle (due to the extra server processes used by Plan B). We consider that it is worth paying the slowdown to buy the extra services provided by the system.

10. Related work

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

Plan 9 [**?**] 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 and much of its code. 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, we have redesigned important system services with portability and adaptation in mind. Besides, Plan 9 lacks a mechanism similar to the constraints we use to select which resources to use and lacks services like context or location.

Some systems permit flexible access to network resources, such as Odyssey [**?**] and Khazana [**?**]. Although some of them consider disconnected operation and adapt to changes in the connection status of the client machine, it is not clear how they can adapt to other changes in the environment, i.e., changes in the availability of a certain device or service.

Unlike Plan 9 and Plan B, none of the systems mentioned tried to use file interfaces as the primary interface for all the system services. None of the systems mentioned below did either.

Systems like the the Semantic File system [**?**], Gaia's Context File System (CFS) [**?**] and the name service in Globe [**?**] 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. Using general purpose tools is harder for them than it is for us. CFS is closer in that it provides a file interface to perform context based queries. We use files for *all* system services instead.

Globe [**?**] and many other systems, like Speakeasy [**?**], Ninja [**?**], Gaia [**?**, **?**], IWS [**?**] and One.World [**?**] 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 application to access the new services just by using the file system interface. For example, Gaia had to introduce a scripting tool [**?**, **?**] and a programming language [**?**] to simplify the use of their system, we can simply use the OS shell. Ninja (whose architecture for services is called SEDA) and One.world are designed to provide services by interconnecting small special-purpose devices through the internet. Although Plan B considers that there might be many small devices exporting services, it has been built as a general purpose computing environment.

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

11. Conclusions and future work

In this paper we have described an approach for building systems to support ubiquitous computing without using any middleware. We have described the system we built, and its most important mechanisms, as well as the design of important system services and several usage examples.

Plan B is the first system that we used that made us think that all the machines *and* the environment are indeed a single computer. 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, sensors and actuators, etc).

There is much yet to be done. Servers to export drivers from other systems into Plan B will be developed as they are needed. The system should undergo fine tuning, to determine the precise reasons for the increase in the system load and reduce it. Peer-to-peer security is ongoing work.

References

[1] Plan b web site., 2001.

[2] *CHX5 The Next Generation Ultrasonic Positioning*. Hexamite, Engadine, Australia, 2005. http://www.hexamite.com/hx5c.pdf.

[3] Plan b user's manual. third edition. *Laboratorio de Systemas, URJC. GSYC-TR-2005-04. Also at http://planb.lsub.org/sys/man.*, 2005.

[4] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. *Symposium on Operating Systems Principles*, 1999.

[5] 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-TR-2004-05*, 2004.

[6] F. J. Ballesteros, E. M. Castro, G. G. Muzquiz, K. L. Algara, and P. de las Heras Quiros. A new network abstraction for mobile and ubiquitous computing environments in the plan b operating system. *Proceedings of the IEEE WMCSA*, 2004.

[7] 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.

[8] 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.

[9] C. K. H. a. R. H. Campbell. A context file system for ubiquitous computing environments. *Technical Report No. UIUCDCS-R-2002-2285 UILU-ENG-2002-1729*, July.

[10] J. Carter, A. Ranganathan, and S. Susarla. Khazana. An Infrastructure for Building Distributed Services. In *Proceedings of ICDCS'98*, Amsterdam, 1998. IEEE.

[11] E. Chan, J. Bresler, J. Al-Muhtadi, and R. Campbell. Gaia microserver: An extendable mobile middleware platform. *Proceedings of 3rd IEEE Intl. Conf. on Pervasive Computing and Communications*, pages 309–313, 2005.

[12] D. P. S. David Garlan. Project aura: Toward distraction-free pervasive computing. *IEEE Transactions on Pervasive Computing*, (2):23–31, 2002.

[13] W. Edwards, M. W. Newman, J. Sedivy, T. Smith, and S. Izadi. Challenge: Recombinant computing and the speakeasy approach. *8th ACM Mobicom*, 2002.

[14] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O. Jr. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 16–25. Association for Computing Machinery SIGOPS, October 1991.

[15] S. D. Gribble, M. Welsh, R. Behren, E. A. Brewer, D. E. Culler, N. Borisov, S. E. Czerwinski, R. Gummadi, J. R. Hill, A. D. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Y. Zhao. The ninja architecture for robust internetscale systems and services. *Computer Networks. Special issue on Pervasive Computing*, (35), 2000.

[16] R. Grimm and B. Bershad. Future directions: System support for pervasive applications. *Proceedings of FuDiCo*, 2002.

[17] H. J. S. I. Kuz, M. Steen. The globe infrastructure directory service. *Computer Communications*, 25, 2002.

[18] B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing Magazine*, April 2002.

[19] T. J. Killian. Processes as files. *Proceedings of the Summer 1984 USENIX Conference*, pages 203–207., 1984.

[20] A. S. T. M. Steen, P. Homburg. Globe: A wide-area distributed system. *IEEE Concurrency*, 1999.

[21] 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 SOSP*, 1997.

[22] S. Nylander, M. Bylund, and A. Waern. Ubiquitous service access through adapted user interfaces on multiple devices. *Personal Ubiquitous Computing*, 9(3):123–133, 2005.

[23] R. Pike. Plumbing and other utilities. *Plan 9 User's manual, Vol 2*, 1995.

[24] R. Pike. Acme: A User Interface for Programmers. *Plan 9 Programmer's manual, 3rd ed.*, vol. 2, 2000.

[25] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[26] 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.

[27] A. Ranganathan and R. Campbell. A middleware for context-aware agents in ubiquitous computing environments. *ACM/IFIP/USENIX Intl. Middleware Conference*, pages 16–20, 2003.

[28] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. Campbell, and M. Mickunas. Olympus: A high-level programming model for pervasive computing. *Proceedings of 3rd IEEE Intl. Conf. on Pervasive Computing and Communications*, pages 7–16, 2005.

[29] M. Roman. *An Application Framework for Active Spaces*. University of Illinois at Urbana-Champaign, 2003.

[30] 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.

[31] uPnP Forum. Understanding upnp., 2004.

[32] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services For Wide Area Applications. In *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, July 1998.

[33] M. Weiser. The computer for the twenty-first century. *Scientific American*, September 1991.