The Design and Implementation of Plan B 3rd edition. A dynamic distributed computing environment.

Francisco J. Ballesteros, Katial Leal, Gorka Guardiola, and Enrique Soriano
7/27/2004
Laboratorio de Sistemas — Universidad Rey Juan Carlos
`http://lsub.org/who`
Madrid, Spain.

ABSTRACT

The 3rd edition of Plan B is implemented by changing a Plan 9 system and adding the Plan B technology to it. We changed the system to make it use volatile late bindings by adding several library functions and a dynamic mount table. Also, new system services have been added to support a highly dynamic distributed environment. In this paper we describe the design and the implementation of Plan B 3rd edition.

1. Introduction

The operating system we use for daily work is Plan 9 from Bell Labs [11]. 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 [16], 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 [10].

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 point provides for customizability of the environment.

When Plan 9 was developed, environments were not as dynamic as they are today. Plan 9 is designed for a rather static environment as shown in figure 1. Users access the system through terminals, that mount files from file servers and may use remote execution at CPU servers. Since devices are exported as files, many resources can be used easily from a remote location.

The problem with this system is that it gets hard to manage when the environment is highly dynamic. Today, our applications usually import many resources from the environment, and the set of resources imported can heavily change during time. This is

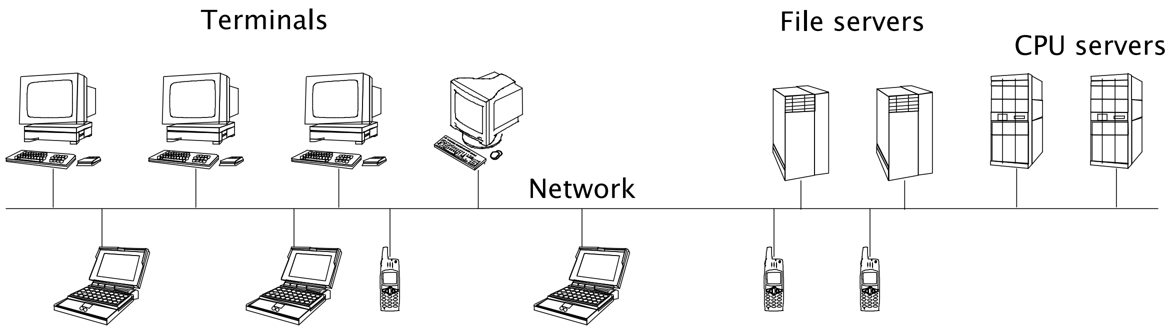

Figure 1: *A Typical Plan 9 environment in the 90s (above the network) and today (also below).*

obvious if we consider what happens when we introduce into the environment new devices like laptops, PDAs, mobile phones, sensors, and actuators[1]. Plan B [1] is an attempt to build a system designed for dynamic distributed environments to provide a convenient computing environment. It builds on both the Plan 9 ideas [10] and our previous research [2] to do so.

Before the current edition of the system, we developed two previous ones that did run hosted on a Plan 9 system. The first edition of the system was completely operational. The second edition had a working kernel and most of the services needed to become operational. This edition was abandoned in favor of the 3rd one, which can be used for daily work. We noticed that an approximation of a Plan B system is a Plan 9 system where

1. resources are exported through high-level interfaces, but also using a file interface; and
2. resources are imported dynamically from the network according to their properties and the application requirements.

We decided to change the system in a way that would preserve the system interface of the original Plan 9 system. In this way, all the Plan 9 applications can be used on our Plan B system.

In what follows, section 2 describes the design guidelines of Plan B 3rd edition, and the rationale behind the changes made to Plan 9. Section 3 shows the overall system architecture. Section 4 describes the new volume service. Section 5 describes the changes to the mount table. Section 6 discusses how to build applications on Plan B and several new library functions. Section 7 comments on related approaches, and section 8 concludes.

2. Plan B 3rd edition

Plan B has relied heavily on the Box abstraction [2]. In Plan B, all resources are exported through boxes, which are a replacement for files. The three main differences between boxes and files are the lack of file descriptors, the use of user-defined attributes, and the unification of the file and directory concept into a single abstraction. Perhaps surprisingly, the 3rd edition of the system does not use boxes, it uses files instead. However, the system has been changed to keep most of the benefits that boxes brought to the previous editions.

[1] As an example of sensors and actuators you may consider X10 devices, that we use to control power sources, lights, and motion detectors.

2.1. Boxes

Unlike files, boxes do not provide descriptors to them, they are handled by using names. This means that Plan B uses volatile late binding, where a resource is bound to a name just for a system call. Once a call completes, the binding is forgotten. The rationale is that if the application does not keep bindings to resources, the system is able to adapt. Adaptation happens because a single name (as supplied by the application) can now refer to different alternative resources. For example, a name for an audio output device can be resolved to different devices depending on the state of the environment.

Boxes have constraints, or attribute–value pairs, that define their properties. The system handles constraints as text, and does not impose any meaning to the set of attributes used. This mechanism is used by the system to select which resources to use for each call. The Plan B system calls included a constraint along with each name, and the resource used had to present a compatible set of properties.

All boxes include a `list` operation, that indeed converts all boxes into both files and directories. A box can be handled as a file by copying data to or from it, but it can be also used as a directory by listing its contents and then performing operations on inner boxes.

2.2. Files used as boxes

We have tried to keep the first two properties of boxes, and have dropped the third one in favor of backward compatibility with Plan 9, ie. we use files and directories.

In our current system, applications use files and there are file descriptors. However, our applications keep files open just to read or write their entire contents. They do not keep files opened just to use them in the future. Two new library functions, `readf` and `writef`, retrieve an entire file and rewrite it using its name as a parameter. The net effect is very similar to a volatile late binding. Each time an application operates on a resource, it uses a name to open the file that represents the resource. Instead of keeping the file open, the file is closed right after reading its contents or updating them. At each open, the underlying system may select a different resource for the same name.

Instead of modifying the file metadata to add constraints, we have added constraints to mounted file systems. In Plan B 3rd. edition[2], services are provided by resource volumes. A resource volume is a file system that is announced to the network and has a set of attribute/value pairs to describe the properties of its resources. For example, one volume may export binaries that are announced to be for Plan B and Intel PC platforms, another may do the same but for Strong Arm ones. As another example, one volume may offer a graphical interface service announced as having a big color screen, another may do the same but could be announced as a tiny screen.

Applications can mount dynamic volumes from the network, requiring them to present certain attributes and to have certain values for them. The result is that the system can change dynamically the name space on behalf of the application. For example, an application may request the system to mount at `/bin` any volume available that contains Plan B binaries for its hardware architecture. It may further request that volumes with interpreted binaries be mounted after the previous ones. In this case, if a native binary is available for the given platform, it would be the one used; an interpreted binary would be used otherwise—should it be there.

[2] In what follows we always refer to this edition.

2.3. Distributed system services

The system assumes that it is common to switch from one resource to another due to environment changes. For example, an editor running at a laptop would be using its disk when disconnected but would use our file server while at the department. Also, an application may use different screens or different keyboards for its user interface. The set of devices used is likely to change during time.

We include volume servers for traditional system services like process execution, authentication, graphics, audio processing, storage, etc. Most of the services are kept as they were on Plan 9. They are repackaged as volumes and announced to the network. However, some services like process execution and graphics have been redesigned and rebuilt to better fit our dynamic distributed environment. We had to redesign those services that were not portable to the wide range of devices that we have now.

For example, the Plan 9 graphics device works well to use workstation-like graphic displays across the network, but it can not be used to operate on mobile phones displays, nor to use displays supported by UNIX and Windows systems. We have built a high-level, portable, distributed user interface service [8] that is better replacement and can be used with our resource volume service.

3. System architecture

The structure of the system can be seen in figure 2. The system is not built upon machines but upon resource volumes. Machines and devices export their resources packaged as resource volumes. A resource volume is a file tree together with a volume name and a description of the attributes for the resource represented by the tree. The attributes are represented and handled as [1] shows, using the implementation of the previous edition for the system.

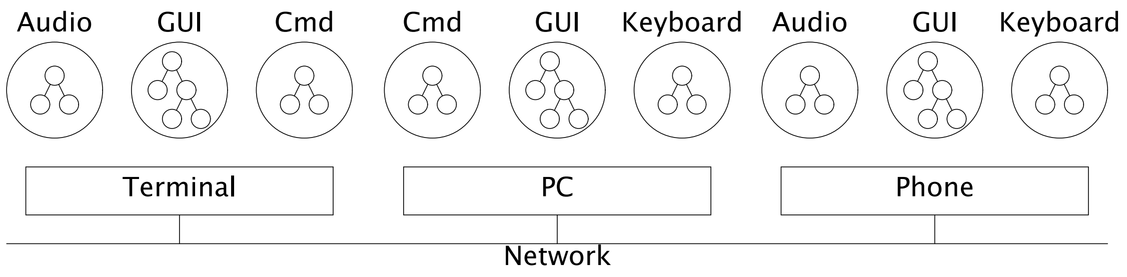

Figure 2: *Plan B: The system is built out of individual resources found in the network.*

Note that resources do not need to be supplied by Plan B systems. Any system may export volumes for us given that it speaks the protocols we understand. For example, the phone in figure 2 may be running Symbian. In the same way, Plan B may export resources to other systems by exporting its files for them.

Together with resource servers, there are resource announcers. A simple announce protocol is used to let the network know of the available resources. This permits client machines to be unaware of which servers may be available. Machines are permitted to move, which means that servers may appear and go during the life of the application that uses them. Therefore, the system includes a volume device to keep track of resource volumes. The system can automatically mount them when they are discovered, and unmount them when they are no longer reachable.

Applications run on Plan B systems. The service to run applications is also a system resource exported like any other service. Such service is provided by volumes for the Cmd service. In the figure, each file shown within the Cmd resource volumes may be a running program that uses the other volumes to perform its job.

Each application has a mount table that supports dynamic volume mounts. A dynamic mount is like a conventional mount operation, but it does not mount a particular file tree, it mounts those volumes available that are of interest for the mount point. The mount request includes a volume name and a volume constraint that specify which properties and values must be met by a volume to consider it of interest. The set of volumes mounted will change during time, and new ones will be added to the mount point as they become available. Those unreachable will be removed from the mount point as they are known to be gone.

4. The volume service

The volume service permits nodes to export and import resource volumes. A volume contains a file tree that provides the service and can be mounted remotely using the 9P protocol [16]. Along with the file tree, a volume has a name and a constraint made of attribute/value pairs.

Constraints are used to determine which ones of the volumes available for a given mount point are of interest for the application. The constraint is a set of values for attributes that refer to properties of the resource volume.

Each attribute/value pair is specified by a rune that names the attribute together with a string that specifies the value. For example, “`Ttext`” is a constraint value for the attribute “`T`” (type) that specifies that the type for the resource is “`text`". Different attributes can be specified together by separating them with the “`!`” character. For example, “`Tbin!Apc`” specifies that the type (“`T`”) for the volume is “`bin`” (binaries) and also that their architecture (“`A`”) is “`pc`” (Intel-PC). See [17] for a description of conventions regarding naming of attributes and their values.

The Plan B `mount` system call permits the application to mount at a given mount point those volumes whose names and attributes match the ones given as arguments.

4.1. Advertising protocol

Volumes are announced and discovered using *vold*, a user-level program that implements a simple announce protocol. *Vold* maintains a set of known volumes by listening to announces. It also announces the volumes serviced from its machine. An announce is a message that contains the following elements

- **Domain:** An administrative domain used to limit the scope of volume names.
- **Address:** The network address of the 9P server that exports the file tree for the volume.
- **Server name:** The name for the resource as known by the 9P server. Usually it is a path to the root directory for the service within the server.
- **Name:** The volume name, used to know what resource is being exported.
- **Constraint:** The attribute/value pairs for the volume, to know the properties for the resource.

The set of announces for each machine is usually configured statically by the machine administrator, although the set of machines can change dynamically. An example of configuration can be as follows:

```
# domain   address            server name        name              constraint
ls         tcp!nautilus!9fs   /usr/nemo          /usr/nemo         !Tdir
ls         tcp!atlantis!9fs   /other/nemo        /usr/nemo         !Tdir
ls         tcp!aquamar!9fs    /dev/volume        /dev/volume       !Tpcm
```

As it can be seen, a machine can announce volumes on behalf of another, when it is not convenient to announce the resources from the machine that services them.

For each new volume seen, *vold* performs the following steps:

1 Dials the file server for the volume and mounts it. The mount point is `/vol/`*mnt*, where *mnt* is the volume name with characters "#" and "/" changed to "`:`".

2 Informs the kernel of the new volume, by writing to `/dev/vols` a string that contains the path for the mounted volume, its name, and its constraint.

When a volume is no longer seen (as detected by the protocol), *vold*:

1 Unmounts the volume.

2 Tells the kernel that the volume is garbage collected, by writing to `/dev/vols` the string mentioned above with `del` instead of a mount point name.

The protocol is very naive. It broadcasts the known volumes at startup and each minute. A volume that is not seen for two periods is consider as gone. When a new machine is booted, or arrives to a new network, it broadcasts an update request that is replied with the announces of those that see the request. The announce messages include a domain name that is used to listen only for volumes of a particular administrative domain.

Announce packet

`vols:\n`
`dom1 addr1 sname1 name1 cnstr1\n`
`dom2 addr2 sname2 name2 cnstr2\n`
...
`domn addrn snamen namen cnstrn\n`
`\0`

Update request packet

`update:\n`
`\0`

Figure 3: *Message formats for the resource volumes discovery protocol.*

The message format uses strings and two different packets, as shown in figure 3. The packet used to announce volumes contains the string "`vols:\n`" and then a variable length string with one line per volume. Each line contains the domain, address, server name, name, and constraint for the volume announced. The packed used to request volumes to be announced now contains the string "`update\n`".

It is obvious that our current implementation would not span several subnets (in the Internet sense). To do so, we could either implement proxies to announce volumes from a different subnet, or implement a hierarchical volume announce service (where the leaves of the tree would be the current implementation).

4.2. Devvol: The volume device

A new device, *devvol*, keeps the kernel aware of the volumes known. The device exports a single file, `#∧/vols`, that is conventionally bound to `/dev/vols`. Writes to `/dev/vols` are usually performed by *vold*, as said in the previous section, to update the kernel idea of the set of volumes available. Reads are usually performed by users, to know which volumes are available.

The device maintains a `Vols` data structure with the set of volumes, and three functions that can be called from somewhere else in the kernel: `volepoch`, `getvols` and `putvols`. The data structure is shown in figure 4. It contains a RW lock to synchronize access to data, an `epoch` number, the number of volumes (`nvols`) and a pointer to an array of volumes. The epoch is incremented each time a new volume is added, and each time a volume is removed. It is useful to identify a point in time regarding volume processing.

Each entry in the `vols` array points to a `Vol` structure, which maintains the information for a volume. New structures are allocated and placed in the array as new volumes are notified by a write to `/dev/vols`. When the user notifies a volume deletion

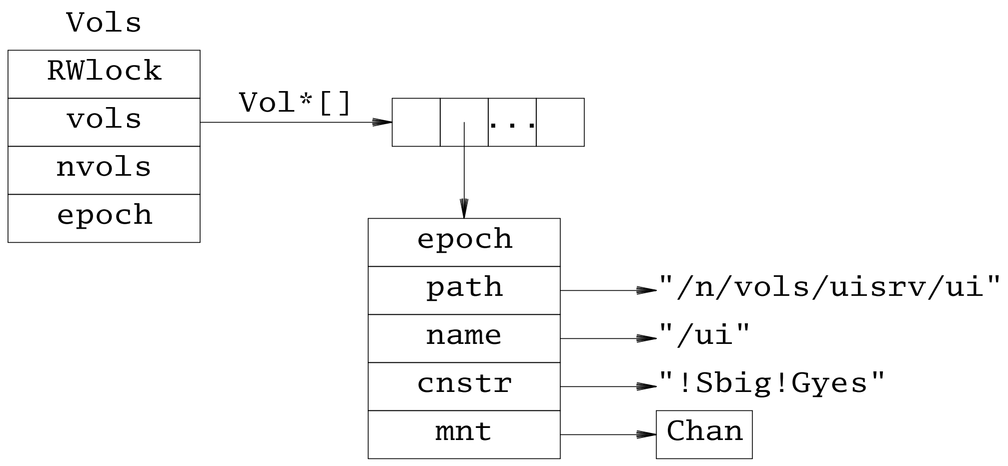

Figure 4: `Vols`. *The data structure of the volume device.*

(also by a write to the same file), the structure is unreferenced and its entry in the array is set to nil. All data structures are reference counted, since they can be referenced from several places in the kernel.

The `Vol` structure includes the value of the `epoch` number in `Vols` at the time of its creation. Therefore, the `epoch` field can be used as an unique id for the volume. The name of the volume is kept at `name` and its constraint is kept at `cnstr`. Both values come from the information supplied in the write to `/dev/vols`. The remaining fields, `path` and `mnt`, correspond to the name for the mounted volume and to the `Chan` for its root directory.

Only this driver needs to hold a write lock on the `Vols` structure. The rest of the kernel may need to acquire a read lock, but it would never write the volume information. All the system calls that resolve names require a read lock, thus there can be high contention on the read locks. This is the reason for using a `RWlock` instead of a cheaper one.

Volumes are mounted on the name of the user who performed the write to `/dev/vols`. This means that the driver is intended for terminals, and not for CPU servers. Using it on CPU servers would require to add a different `Vols` structure for each user. Although the code is written taking that into account, there is only one structure declared in the driver for the moment.

The kernel can check which one is the current epoch for volumes by calling the `volepoch` function, which returns the value of its `epoch` field. That is useful to know if another data structure is up to date regarding volume processing, by storing the epoch in the client data structure and comparing it later with the current epoch number. The idea is similar to the use of version numbers to detect out of date cached data.

There are two functions, `getvols` and `putvols`, that permit gaining safe read access to the volume information. The first function acquires a read lock and returns a pointer to the data structure. The caller can perform whatever processing requires read access to it. Once the caller is done, it must call `putvols` to release the read lock. The next section shows how the name space implementation uses this mechanism to update the mount tables for processes using volumes. It should be noted that while the lock is held, the volume information is not updated. Any process writing to the volume device would block trying to acquire a write lock on it.

5. Dynamic mount tables

We describe now the implementation of the dynamic name spaces in Plan B, but we focus just on the changes made to the standard Plan 9 mount table implementation. A complete description of how the Plan 9 mount table implementation works is out of scope in this paper, you can refer to [3] instead.

The mount table is implemented by a `Pgrp` structure. Each process has a pointer to a `Pgrp` that might be shared with other process. The structure contains a hash table for mount headers, or `MHeads`, each one representing a mount table entry. The entries are hashed using the mount point `Chan` as the key for the hash function[3]. For each hash value, the different entries in the table are linked through the `hash` field. This can be seen in figure 5.

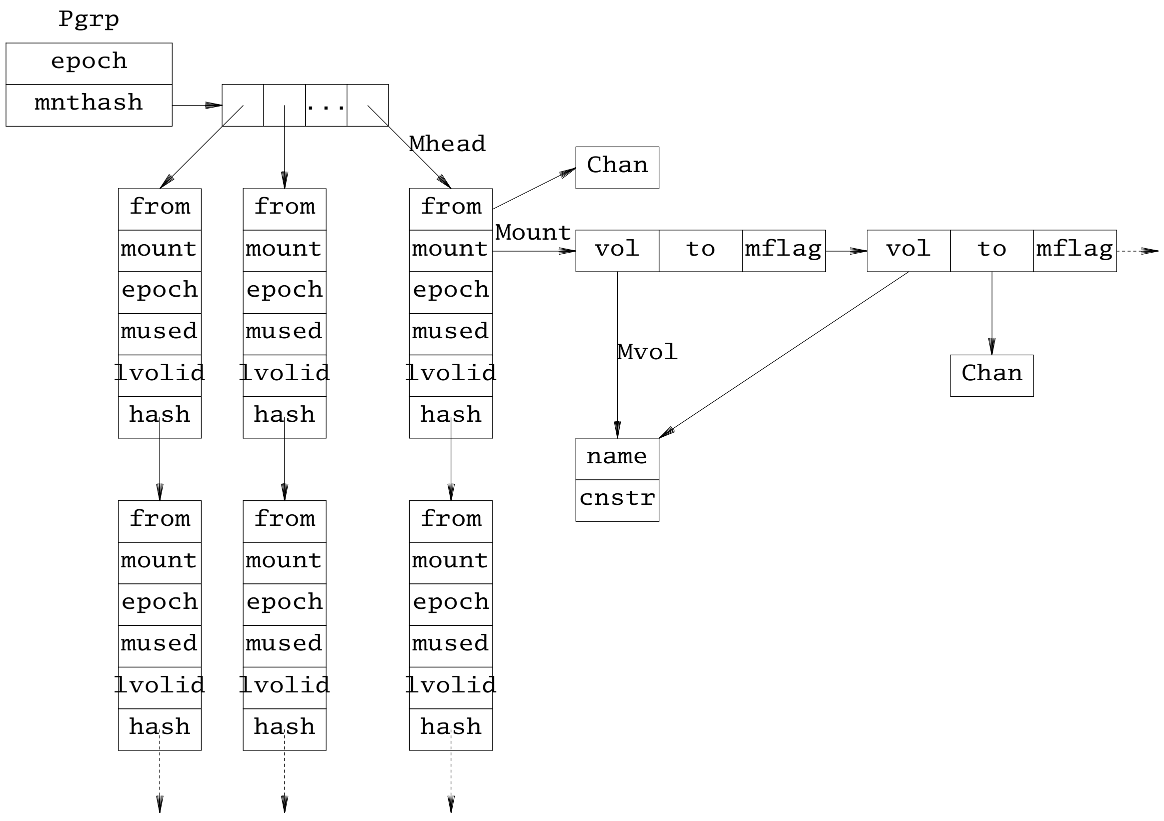

Figure 5: *The implementation of the dynamic name spaces.*

A mount point is described by the `from` field of its mount table entry, which points to the `Chan` for the file that is indeed the mount point. When a name is resolved, the different `Chans` obtained at each step of the path resolution are looked up in the hash table. When the `Chan` is found in the table, it means that the user has mounted something on it. The kernel crosses the mount point by looking at the `mount` field, which is a list for the files mounted on it.

Each entry in the `mount` list is a `Mount` structure that represents a mounted file. Its `to` field points to the `Chan` for the mounted file. In Plan 9, unions have multiple `Mount` entries in the list—other mounts that are not unions have a single `Mount` node. For example, when `$home/bin/rc` is bound after the previous contents of `/bin`, the

[3] The key is actually the `type`, `dev`, and `Qid` for the channel, since that identifies a file.

list would contain one node for the old /bin file and then another for the $home/bin/rc file. Each node would have a to field pointing to the Chan for the corresponding file. All the nodes have a valid to field that points to a valid file.

The Plan B mount table can mount volumes as well. Each mount entry, ie. each Mhead, has a new epoch field. This field is zero for conventional mount entries, and has a valid volume epoch otherwise. The epoch kept in epoch was the current one at the time of the last update for the Mhead. This is used to know quickly if a mount entry is out of date or not. A mount entry that has an epoch whose value is less than volepoch() may be out of date. When the kernel tries to use an out of date entry it would first (r)lock the volume device and update the entry.

Figure 5 shows how a volume Mhead has a Mount node with a nil to field, since there is no file mounted there. Instead, the Mount entry includes a non-nil vol field that points to a mount volume information (or Mvol). All conventional mount nodes have their vol field set to nil. The mount volume information includes the name and constraint for the volumes of interest for the mount point.

Thus, after a mount request for a volume, the mount table would have an entry with just a single Mount node that holds the Mvol. We refer to this node as the volume insertion point. After the insertion point has been established, the kernel looks at the volume information to automatically mount after it all the volumes that match the given name and constraint. For each volume mounted, a new Mount node is linked. The to field for each node points to the Chan of the mounted file, and its vol field points to the Mvol. The Chan for to is obtained by cloning the mnt field of the corresponding volume in the volume device.

Although volume mount entries look like mount unions, they are not. When there are several alternative volumes mounted at a single insertion point, only one is used. For example, a mount call can request for any volume known as /usr/nemo to be mounted at /usr/nemo. If we have two such volumes (eg. one from our local disk and one from the file server), we would have two different Mount nodes in the mount table entry. Plan 9 would then behave as if the two volumes were unioned at the mount point. However, Plan B uses just the first entry available, and ignores the other ones. This corresponds to the user idea of using one of the volumes available for the resource, but not all of them at once.

The Mount node used is flagged with MUSEVOL (using its mflag field), and other ones are ignored by the name resolution code. As a cache, the mount head includes a mused field that points to the Mount node used. This cache is necessary to recover from volume failures, as said later.

Under certain circumstances, users want indeed to union all the volumes available for a given (volume) mount point. In this case, all the Mount nodes are flagged as MVOLUN. This makes the kernel union all the nodes, as it would do for union mounts.

5.1. Adapting to changes

When a volume being used gets unreachable, and an alternative is selected for use, the mount table is rewritten to use the new one instead. All the mount entries that point to or from the old volume are rewritten to point to or from the new one.

Before using a Pgrp, the kernel calls pgrpupdatevols(). This function ensures that the mount table is not out of date regarding volume processing. If the table is noticed out of date, it is scanned to see which volumes should be mounted and which ones should be unmounted. Any volume mounted that is not found in the volume device is known to be gone, and therefore unmounted. For each Mhead that corresponds to a volume mount, each matching volume with a bigger epoch than the one in the Mhead must be mounted at its insertion point. Once the Mhead is updated, its epoch field is updated with the current epoch. The same is done to the Pgrp once all its mount entries have been updated.

Processing of volume Mheads is not enough to update a mount table. There can be entries that mount files from a mounted volume, or that mount into files coming from a volume. When a volume is unmounted, such entries must be updated. The kernel scans all the mount table looking for entries with to or from fields pointing to the unmounted volume. For each such field found, the Chan is replaced with a new one. The new Chan points to the file with the same server name that comes from the new volume used. Note that an unmount of a volume would make the next one in its list to become the one flagged as MUSEVOL.

Rewriting a Chan needs some help from it. We modified the Chan structure to know which volume (if any) it comes from, and what is the name within the volume for the file. As shown in figure 6, the field volid contains the id for the volume and the field sname contains the server name for the channel. The figure shows a Chan that is pointed to by the file descriptor table (or Fgrp). Note also how channels that do not come from a volume would have its volid set to zero. The redirection of a channel to a new volume is performed by chanredirect(), that receives the old channel and the mnt entry for the new volume. The function clones and walks the root of the new volume as needed to reach the server name for the old channel.

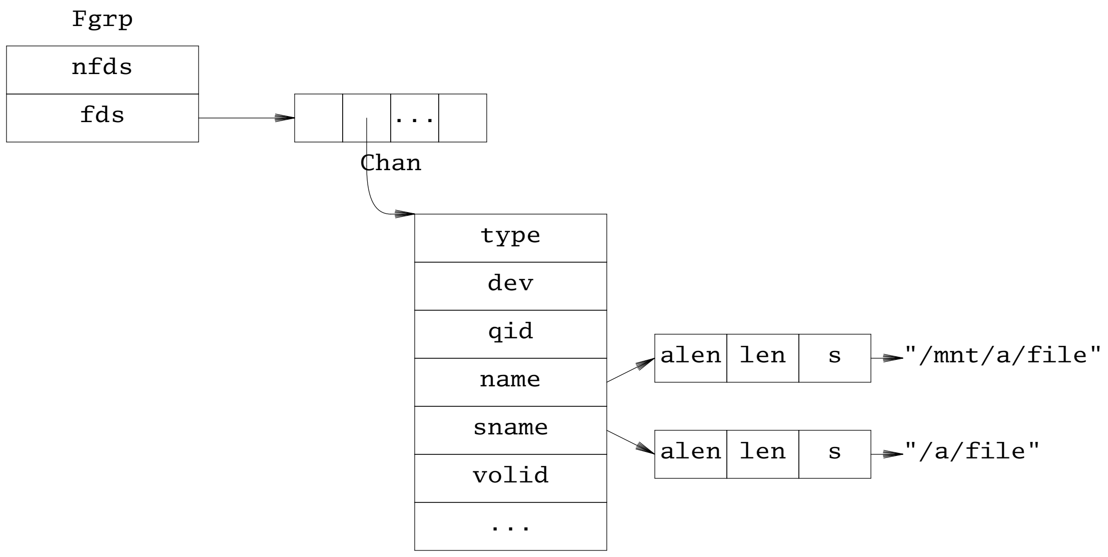

Figure 6: *Plan B channels.*

In some cases, the only available volume for a mount point may be unmounted. In this case, we cannot rewrite the channels to point to the new volume, since there is no one. What we do is to record in the Mhead the volume id for the gone volume (using the lvolid field). When a new volume for the mount point becomes available, all the channels in the mount table using the lvolid volume id are rewritten to use the new volume.

Finally, it must be said that the kernel does not attempt to rewrite the Chans kept by the kernel outside the mount tables. This means that if a process holds an open file descriptor pointing to a volume file, and the volume goes away, the Chan becomes broken. Any further operation on the descriptor will fail. This is useful to let the application know when a file becomes unavailable while it is in use. Should the application care, it is expected to reopen the file and retry the operation. By reopening it, a different volume may be selected for the given name by the mount table. It becomes now clear why we strongly advise our applications not to keep files open. The next section discusses how this issue is handled.

6. System interface

The user interface is kept as in Plan 9, however, the name space can now change underfeet when volumes are used. The mount system call includes two new flags MVOL and MVOLUN that can be used to mount volumes and to mount volume unions. The mount user program has been changed to include new V and U flags that correspond to the new system call flags. The mounted volumes are handled as said in the previous sections. For example, this command line

```
mount -aV /usr/nemo!Tdir!Dgsyc /usr/nemo
```

corresponds to the system call

```
amount(-1, -1, "/usr/nemo!Tdir!Dgsyc", MAFTER|MVOL, "/usr/nemo");
```

This call mounts at /usr/nemo any volume named /usr/nemo with constraint !Tdir!Dgsyc (type directory and domain gsyc). Further calls like

```
bind -bc /usr/nemo/bin/rc /bin
```

work as expected. At any time, new volumes matching the request are mounted in the volume mount point, and unreachable ones are unmounted. The first volume found is the one used, so that the user does not see a union. Should the volume used become unreachable, the next one is the one used.

To make it easy for applications to behave well, ie. to avoid keeping file descriptors open, two new library functions are provided:

```
void*   readf(char* file, int* l, int retry);
int     writef(char* file, void* buf, int len, int retry);
```

Readf reads the named file and returns a pointer to its contents. The number of bytes read is returned in *l, if l is not null. The function stats the file to learn its length, and then uses readn to read the entire file contents into dynamic memory. When the file length is reported to be zero, a single read is performed—This happens for stream devices, that return data each time a read is performed. The caller is responsible for calling free on the memory used to read the file, once the data is no longer necessary. As a convenience, the allocated memory contains an extra byte that is set to zero, to allow the caller to use the memory as a string (for text files).

Writef performs the opposite operation. It writes the given buffer into the named file. Both calls keep the file open only while it is being used. The application that uses them keeps the files closed most of the times and uses the file names to get to them.

As a convenience, both functions can be instructed to retry if an IO error is noticed while accessing the file. They do so when the last parameter is non-zero. In this way, if a volume is gone while a file is being accessed, an alternate volume is used in a transparent way.

The combination of the dynamic mount table and the two new library functions, brings to our system the volatile late binding of constrained resources that our previous editions had.

7. Related work

The main difference between our approach and other systems also designed for distributed dynamic environments is that we use the file system protocol as the basis for distribution. In fact, our current system is a descendant of Plan 9 that includes what we learned with previous editions of Plan B to adapt to environment changes.

Plan 9 [11] and Inferno [5] are the most similar systems, but unlike Plan B, they do not adapt to changes in resource availability. Also, services like remote command execution and graphic devices are not portable enough to permit applications to distribute among the set of nodes available. For example, Plan 9 applications can not redirect their user interfaces to whatever display is selected by the user. Our dynamic mount table, combined with a portable server for user interfaces [8] can do so.

There are many other systems, like Ninja [6], Gaia [12], Globe [13], One.World [7], etc. that rely heavily on middleware services as the means to implement and distribute their services. Unlike our system, they do not provide a complete computing environment since their applications require services of the native system underlying their middleware. For example, Gaia uses Windows XP applications to display slides, but, to the best of our knowledge, such applications are not able to operate on displays serviced by a different operating system.

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.

Native systems like Symbian, Windows CE, and Palm OS do not adapt well to environment changes, nor are they designed as distributed environments. Our system is.

Many systems improved distributed file systems to add features like multimedia services [4], disconnected operation [9], etc. None of them tried to use such interfaces as the primary interface for all the system services.

WebOS [14] 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.

Last but not least, one point that shows the difference between our system and related approaches is that ours is both our ubiquitous and mobile environment 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.

8. Conclusion

We have shown the design and implementation of the third edition of the Plan B system. The system relies on a dynamic mount table as the means to adapt to environment changes. Other mechanisms like a volume device, a discovery protocol, and several new library functions have been introduced to merge the pieces together. For our approach to work, *all* system services must be provided using interfaces with a high-level of abstraction, and all of them must be provided using files as the I/O mechanism. We have kept those Plan 9 services that present a reasonably high level of abstraction, and have replaced those services that do not.

The system is still experimental, but can be used for daily work. All the Plan 9 applications run unaware of the new semantics of the underlying system. We are now in the process of redesigning several system services and porting Plan 9 applications to use them. In papers found somewhere else [15] we describe the progress made so far.

References

1. 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, *Submitted for publication. Also in http://lsub.org/ls/export/box.html*, 2004.

2. F. J. Ballesteros and S. Arevalo, The Box: A replacement for files, *Proceedings of HotOS–VII*, 1999.

3. F. J. Ballesteros., Notes on the Plan 9 3rd Edition System Kernel., *GSYC–Tech. Rep.–2000–XX Available at http://lsub.org/.*, 2000.

4. S. Childs, Filing system interfaces to support distributed multimedia applications, *Eighth ACM SIGOPS European Workshop Support for Composing Distributed Applications*, 1998.

5. S. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey and P. Winterbottom, The Inferno Operating System, *Bell Labs Technical Journal 2*, 1 (1997), .

6. S. D. Gribble, M. Welsh, R. Behren, E. A. Brewer, D. E. Culler, N. Borisov, S. E. Czerwinski, R. Gummadi, J. R. Hill, A. D. Joseph, R. H. Katz, Z. M. Mao, S. Ross and B. Y. Zhao, The Ninja architecture for robust Internet–scale systems and services, *Computer Networks. Special issue on Pervasive Computing. 35*, 4 (2000), .

7. R. Grimm and B. Bershad, Future directions: System Support for Pervasive Applications, *Proceedings of FuDiCo 2002*, June 2002.

8. G. Guardiola, The implementation of Plan B's UI server, *GSYC–Tech. Rep.–2004–XX. Also in http://lsub.org/ls/export/uiimpl.pdf*, 2004.

9. 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.

10. 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.), .

11. R. Pike, D. Presotto, K. Thompson and H. Trickey, Plan 9 from Bell Labs, *EUUG Newsletter 10*, 3 (Autumn 1990), 2–11.

12. 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.

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

14. 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.

15. Plan B web site., *http://lsub.org/planb*, 2001.

16. Plan 9 Programmer's Manual, *AT&T Bell Laboratories. Murray Hill, NJ.*, 1995.

17. Plan B User's Manual. Second edition., *Laboratorio de Systemas, URJC. GSYC–Tech. Rep.–2004–04.*, 2004.