Architectural overview of Plan 9

Stijn van Drongelen
María Hernández
Johannes Leupolz
Alejandro Serrano

Contents

1 Introduction

Today most computers are part of a network. Being in a networked environment makes it easier for the users to share data and to communicate with each other. Many modern operating systems like Linux or BSD-derivatives have their roots in UNIX. But UNIX itself was originally not designed for networking - Instead new functionality was just attached to the kernel [Pike91] and integrated with a poor design.

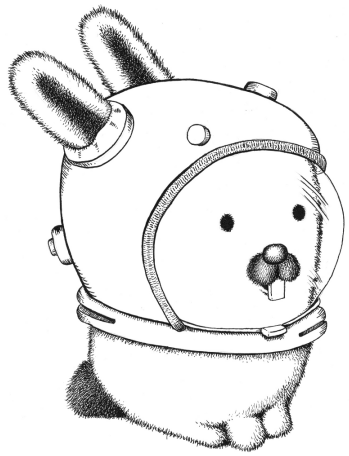


Figure 1: Glenda, the mascot of Plan 9

Figure 1 shows the space bunny Glenda[1], the mascot of Plan 9. Plan 9 from Bell Labs is an operating system developed since the late 1980s in the Bell Laboratories. The same research team started the UNIX development earlier[Pike91]. Eric S. Raymond describes Plan 9 as "...an attempt to do Unix over again, better"[Raymond03]. It was built around the idea that every service that the operating system offers should be accessible in a big file hierarchy.

To describe the architecture of Plan 9 we focus on insightful trade-offs - we expose use-cases around these trade-offs and dive into the system to give the reader a deeper understanding of Plan 9. We use mainly papers provided by the development team of Plan 9 and parts of its source-code. To show how alternative operating systems solve the issues which occurred we compare Plan 9 with Linux and GNU Hurd (also called the Hurd). We have chosen Linux as representative of UNIX and GNU Hurd as an alternative successor of the traditional UNIX with similar ideas but built around the micro-kernel Mach and with a strong POSIX-compatibility.

In the section 2 we give a brief overview over Linux. Afterwards we describe the main principles of Plan 9 to give an *technical overview* over the novel ideas of this operating system. At the end of section 2 we introduce GNU Hurd. In section 3 we have a closer look at quality aspects: First we describe main quality aspects of operating systems in general and then we show on which quality aspects the Plan 9 developers focused on and which aspects retreat into the background.

[1] image taken from `http://plan9.bell-labs.com/plan9/glenda.html`

Section 4 introduces trade-offs of design decisions which where made in Plan 9. To get a better insight of the architecture we dive into the system by having a look at main trade-offs which arise because of *simple and elegant interfaces* in section 5 and *remote transparency* in section 6, respectively. To do so we first describe what exactly is the trade-off. Then we describe its assets and drawbacks and show a use-case where the trade-off arises. To enrich our findings we also have a closer look at the involved source-code and the situation in alternative systems.
To conclude this paper section 7 summarizes the findings and makes a forecasts on the future research around Plan 9. Especially we focus the Plan 9 successor *Inferno* and influences of Plan 9.
Finally the appendix in section 8 gives answers to questions about Plan 9 which arose.

2 Overview

The distributed operating system Plan 9 was started by the mid 1980's by Bell Labs. The same people who developed Plan 9 were the designers of Unix. It started as a research system, the successor of Unix. They wanted to build a new system that fixed or improved some aspects of Unix, as the problem with the big amount of different servers that were created when Unix was spread over the world. But basically, they only wanted to provide Unix with a namespace approach and a file-oriented simple protocol.

Unix is a monolithic system built over the idea of treating the whole system as a unique hierarchy: there is only the root directory, / and from that mount point, the rest of the files were descendant. However, this vision goes much further in Plan 9: it is not only the organization of the system which is treated as a file, but (almost) *every* system calls. Despite of this was the main change, a lot of work had to be done to the kernel, so they decided to start a new one from scratch [Plan9FAQ].

Plan 9 takes some features of Unix, but it also includes new aspects. It is based in the main idea, *everything is a file*, supported with other two important principles which constitute the frame of the system: files are accessed either locally or remote with a standard protocol and private namespaces for each service.

Plan 9 was started by Ken Thompson, beginning the protocol; Rob Pike, who took charge of naming and the C compiler and Dave Presotto of networking and the window system. Also Phil Winterbottom and Dennis Ritchie participated in the project. All of them were members of the group Computing Science Research Center, the same group that developed Unix and C language. Besides, during the development of Plan 9 other people took part into it.

The *first* release appeared in 1992. It was only available to universities. It included the most representative parts of Plan 9, such as the kernel, *sam*, the screen editor, *Alef*, a programming language designed by Winterbottom, and full UTF-8 support.

The *second* edition was launched in 1995. Then, it was made available to the general public, with a cost of $350. The distribution included a license for the full organization. In that, it was added *acme*, a user interface for programmers.

The *third* release was intended to change the name of the system to Brazil, after a reimplementation, but at the end the original name was kept when it was put on shell, in 2000. This edition introduced *draw*, a color graphics operator and *wrap*, an update manager to install packaged updates. In this case, it was distributed for free via the Internet, under the Plan 9 license.

Finally, the last edition, the *fourth* one, was released in 2002. It included many important changes in the system. Actually, a redesign of many parts of the system was made: the file system protocol, 9P was turned into 9P2000, and also the kernel and libraries were modified, either rewritten or rebuilt. This version is a total change with the previous one: it introduced security tools, *factotum*, encrypt connection for some system calls, ... but a very important point is that the Plan 9 protocol over 9P until that moment, IL, was relegated only for being used by *fs*, and all the others communications started to use TCP. This change was introduced because IL protocol did not work

well with long networks [2]

In this last release, besides the previous named changes, a new mechanism for getting updates of the system was built. Nowadays, new revisions are dairy updated and user can just download them. The current tree with the changes is available at the *fossil* server, a public server at `sources.cs.bell-labs.com` [Overview].

2.1 The three principles

The three basics principles of Plan 9 leads the system to have some characteristics which make it different from other operating systems. With that new approach, a service in the system is started with a write operation in a text file. Later, the 9P protocol reads the data written in the file and passes the information to the kernel, which starts other actions in order to finish the request. Instead of having an API or system call, you have a file doing the same job. Files in the system are like objects, it is file-oriented.

As an example, in order to kill a process in Plan 9 it would only be necessary to write `kill` in the file `/proc/p/ctrl` file, where `p` is the process identifier. This `/proc` filesystem stores and gives information about the process, such as its status, the memory it allocates, the devices it is using, the values for the processor registers,....[Ballesteros07] For example, the command

```
awk '{print $1}' /proc/*/status
```

prints the name of all the processes running in that moment.

However, this approach is not extended to every resource in the system, there are also system calls. In the previous `kill` example, it is still necessary the command `fork` or `exec` to run the process. It is not enough with a sentence similar to

```
cp /bin/date /proc/clone/mem
```

The `/proc` file server is only an abstract representation of the process, not the process itself.

Another resource which is not suitable for being file-oriented is the shared memory. Plan 9 provides other mechanisms to let group of users access and modify shared memory. But, in spite of these (and others) examples, the file-oriented structure makes Plan 9 a good distributed system, as we will later see. This approach was included before in some other previous systems, but it was fully developed at Plan 9, getting better results.

Another principle can also be derived from this idea, let's say, Principle 1bis: *most names and contents in the files are human-readable*. Textual forms makes the system simpler to use for the users or developers than using binary files. Services that present file-like interfaces are, usually, easy to understand, and easy to use. Plan 9 was one of the first systems to support Unicode natively.

[2] More information about all these changes at [PikeFourth], but a further reading on the present document is recommended before getting into that one.

The second, but as important as the previous one, principle is about the namespaces [PikeNames]. They are private for each process. It means that, each process running in the system, has its own file tree. The resources form a file hierarchy in the environment. When a new process is created, it constructs a new file name space that attaches to those resources. This is better seen with an example.

The window system, $8\frac{1}{2}$

It is a server for files as `/dev/cons`, `/dev/bitblt`, `/dev/mouse` or `/dev/screen`. The mouse file returns the position of the mouse when it is read: a string with the x and y coordinates. Until here, there is no big difference with the way of proceeding in other systems, but it is in the fact that the `/dev/mouse` file –and others– is a different copy for each service, in its local name space, there is an instance of the file for each client. Each process has its own mouse file, with different data, getting different values when a read call is done.

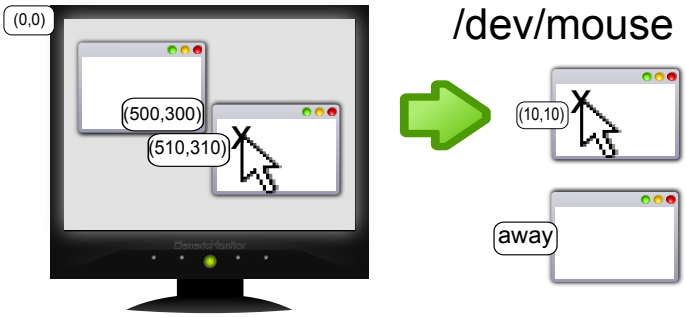


Figure 2: Example of `/dev/mouse` behaviour

As Figure 2 shows, the mouse position is not the same for each window (running in different process). For the main window it is at $(500, 300)$, while for the window below it is at $(10, 10)$ and for the upper one it is not even inside it. The mouse is actually in only one place, but each process considers that it is the only one running and has its own view of the mouse.

`/dev/cons` has the same behaviour. Each client holds a different view of the console. The file-oriented procedural in Plan 9 simplifies the task of defining the standard input and output of a process, with respect to the way of doing that in Unix. In it, special operations must be made in order to specify the behaviour of `/dev/tty`, with operations involving the kernel, whereas only write operation at `/dev/cons` indicating the standard input, output and error files – similar to the ones for accessing bitmaps graphics– is needed.

An finally, the third principle, *transparent remote access*. It means that Plan 9 doesn't distinguish whether a file which is being accessed is in the local machine or in a remote one. It only needs to know the file system hierarchy for that file. If the file is remote, it constructs a name space in the local machine connecting the file.

The access to remote files is performed using the 9P protocol (changed to 9P2000 in the last release). This is a network-level protocol and it uses IL, a protocol built on top of IP, that provides the error-correction and packages facilities that 9P needs. The

protocol is file-oriented and contains 17 message types. 9P messages are generated by the kernel after an I/O request, either started by a user or by the kernel itself.

A connection to a piece of name space in a remote system to the local one is started with the `import` command. It starts a local process with the remote file hierarchy. From that moment on, the remote files are accessed as if they were local to the machine.

A good example of this situation is the *remote debugging*. One can access the processes on a remote machine by mean of the command

```
import helix /proc
```

It makes processes at Helix machine (with helix being a central server) available at the local machine. Instead of seeing the local ones, only the remote ones can be seen (it could also be possible to see both, but this command overwrite the local programs). Now, one can debug processes running on helix as if they were running locally. It is also possible to see the status of the processes, the memory they are using, etc. The debugger could now debug those processes:

```
db /proc/1/text /proc/1/mem
```

Another example is

```
import helix /net
```

Helix is a machine with a lot of network interfaces. After this command the local machine has access to those interfaces, even it was initially only connected to Helix, and can run services in those networks.

There are some commands in Plan 9 related to remote connection. The `import` command, seen at the example, but also `exportfs` and `cpu`. The first one is just the necessary thing that a server has to do in order to make available over the network for a later remote connexion. It allows other machines to attach the file server, by exporting a portion of its own name space. When an `import` is required, it starts (if allowed by authorization) an `exportfs` in the remote machine. The `cpu` command works in the opposite direction to `import`. It attaches one portion of the local file system to the remote server. After that, a shell can be run in the server, which will probably be faster than the machine.

This procedure allows users to create their environment and recreate it later on at every machine, whenever they want, just with a cpu connection or import command.

Plan 9 has also a new concept, *union* folders. They are points in the name space where some directories are mount in that same point. For example, after running

```
import Alice/proc /proc
import Bob/proc /proc
```

both `/proc` directories from Alice and Bob are mounted as if their processes were local. After doing that, all their processes can be accessed. But the relevant point here is: How is the `/proc` directory managed? It means, for example, if we type `..` at that directory, where will we go? If we create a new file, where is it actually going to be?

The first question is solved with just a default rule: union folders are viewed as a rear, made up of the directories in the union; the first directory mounted in the name space is at the front of the rear; the next ones are just concatenated, added at the end. This behaviour can be explicitly changed in the `mount` call, specifying the position of the new added directory. The second one has also a default rule: directories which are unions do not accept new files in them. It can also be modified with the `create` system call applied to the file. The file is created in the first directory in the union with create permission (this permission is given with a flag in the `mount` or `bind` call). These and other technical concerns about union files are exposed more deeply in the section about the first important trade-off.

2.2 The Plan 9 approach

The three principles together set up the Plan 9 architecture and its particular approach:

$$\begin{array}{rr} & \text{Everything is a file} \\ \wedge & \text{Files can be accessed everywhere} \\ \hline \vdash & \text{Everything can be accessed everywhere} \end{array}$$

This is actually the big point of Plan 9, and the aim that its authors wanted to get. It has been built over the idea of a distributed system, with the services spread out over more than one computer. If everything can be accessed everywhere, why should Plan 9 users have everything installed in their local and private machines? It is not necessary anymore. A normal installation of Plan 9 consists of several computers, with a service allocated in each. On this way, services which require fast computation can be installed in fast machines. Private machines just only have to provide access to those computing and file servers.

Those servers are connected together by the network. As they are intended to run applications whose results are going to be used by others, they are usually connected by high bandwidth networks, becoming a fast local-area network, see Figure 3[3]. Home or office workstations or PCs are connected to this network by lower bandwidth wires, such as Ethernet or ISDN. They are called terminals.

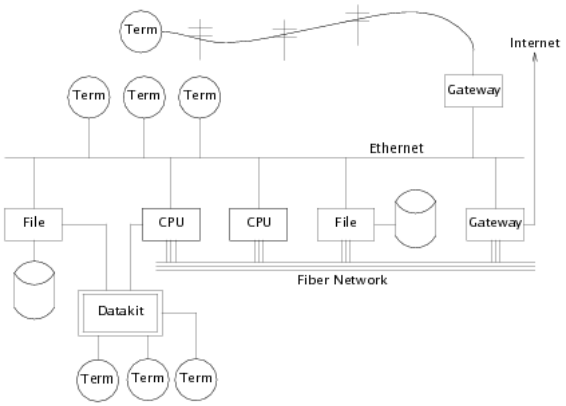


Figure 3: A typical large Plan 9 configuration

[3]image taken from [Pike91]

Each server provides a specific task. They export their file hierarchy, making the services available over the network. Users can make their system as its own desire, just choosing which services they are interested in, and they ignore the ones they don't need. This is particular form of configuration, instead of buying or installing programs locally, they are remotely accessed, and with a better performance as they are established in fast machines, improved for that task. Let's see another representative example.

The remote soundcard

Let's think about all these concepts and put them together in order to exploit some of the powerful of the principles. With Plan 9 it is possible to use a sound card in a remote system as if it was your local sound card and play music in that computer.

Alice is the owner of the sound card at her computer, and Bob doesn't have, but he wants to listen to his favourite song. He can run

```
export /dev/sound Alice/dev/sound
```

If he has access to Alice machine, that command will success and he will have just to play the song, and it will sound at Alice's computer, see Figure 4.

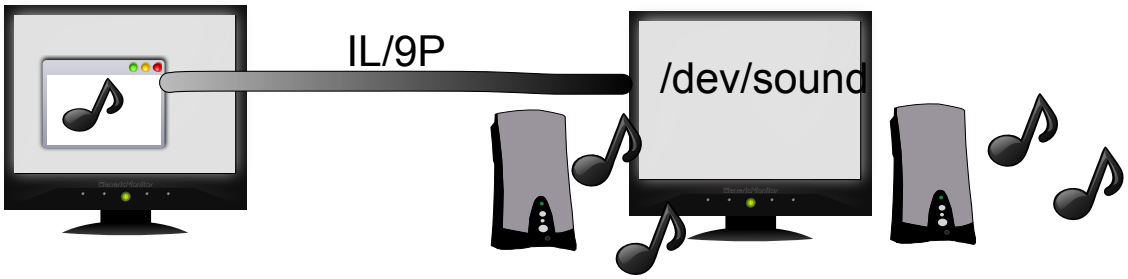


Figure 4: Example of the use the 9P/IL protocol to play remote music

2.3 Other components of Plan 9

In the previous subsections we have already exposed some programs in Plan 9, but there are also some other basic software and protocols that must be known to get a better view of the system.

- alef: a C-like programming language with concurrence support. It was abandoned and a thread library for C was written, due to the problems with the number of people working at the same time in the system and the difficulty of maintain it across multiple architectures,
- rio: the window system of Plan 9, a complete rewrite of $8\frac{1}{2}$ in Alef – later rewritten in C,
- acid: the Plan 9 debugger,
- IL: a protocol over 9P (or later on, over 9P2000) which deals with security and error. Nowadays it is actually mostly unused, in detriment of TPC,

- 9P2000: a new protocol introduced in the fourth release that removed some restrictions on name lengths and introduced authentication files as a mechanism for moving the details of authentication protocols out of 9P, among others,
- Factotum: the security agent, introduced in the fourth release,
- Secstore: the key store, introduced in the fourth release,
- Replica: a tool for getting the new revisions of the system,
- ...

2.4 Hurd

Hurd is a multi-server operating system built over the Mach microkernel. It started being developed in 1990 by Thomas Bushnell, its initial architect. GNU Hurd appeared as part of the GNU project, the kernel was the most important missing component in the way to create an operating system consisting only of free software [Walfield07]. They chose a microkernel since they considered it would lead to get better results than with the traditional Unix monolithic kernel architecture, at least in some aspects. However, its first option was not the GNU Mach microkernel, but the 4.4BSD-Lite kernel, but after some discordances with the company, they will opt for Mach.

The aim of creating a new kernel was to rectify some observed shortcomings in Unix. They also wanted almost total compatibility with the Unix-like kernel and a focus on simplify sharing and distribution. The choice of a microkernel, a multi-server architecture and a user extensible framework allowed all of their desires [BrikmannHurd]:

- the Mach microkernel is POSIX compatible and doesn't give up the Unix security model,
- the extensible framework allows the user to configure the system, but not only the appearance or the programs they want to use, also the filesystem format or the network protocol, among others; a desirable requirement of free software,
- the microkernel leads to a more stable kernel, as there are less code to break,
- in a multi-server system, each server program is responsible for a unique service. This provides more safety to the system, as if one server breaks down, the others still alive and can be used. Also the system can evolve easily by adding new servers to it.

The Hurd system defines interfaces, each of them providing a service. The system is object-oriented, and objects implement an interface, extending the functionality of the system. These objects are implemented in servers, user-space processes. Usually, these servers don't hold a whole object, large objects are spread out over some servers. The way of communication in Hurd are *capabilities*, which designate and object and authorize access to it. There are about a dozen of interfaces in Hurd, such as `fs`, `io`, `fyss`, `exec`, `process` or `auth` and `password`. The reference to the object is made by a

message in a kernel message queue, the *port*. By mean of this capability a process can read or modify an object. As typically, a client enqueues messages in that queue and the server dequeues them. For that, they must hold some capabilities: the *send right* for the client and the *receive right* for the server. They use the RPC – Remote Procedure Call–: the client usually also includes a *reply capability*, designating the queue where the messages are stored.

We have chosen Hurd as the system to compare with Plan 9 because it follows a similar point of view: instead of files in Plan 9, Hurd is built around messages. Also, it has a distributed architecture, the same idea that Plan 9 gets with the file serves distributed in different computers.

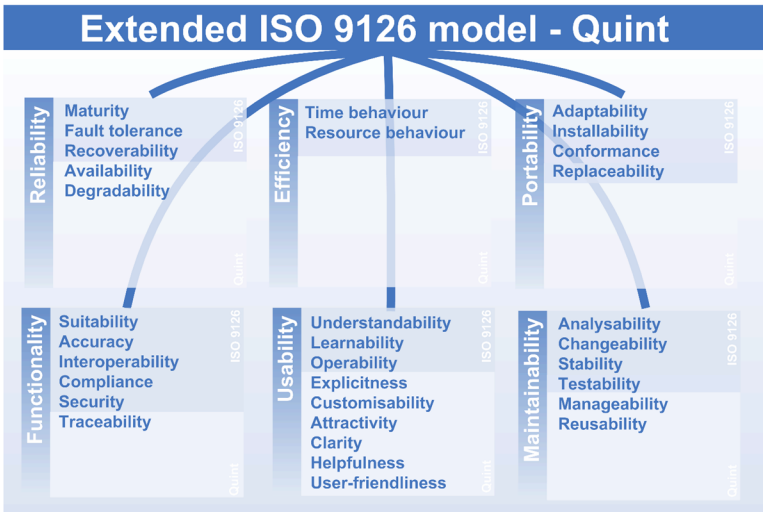


Figure 5: Quint2

3 Quality aspects

3.1 Quality standards

ISO 8402 defines quality as "the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs"[Bevan99]. The ISO 8402 defined basic concepts and the language of *quality*. The standard was replaced in the year 2000 by the ISO 9000 family of standards[ISO08].
In ISO 9000 "quality of something can be determined by comparing a set of inherent characteristics with a set of requirements"[Praxiom10]. For our analysis of Plan 9 we use characteristics defined in ISO/IEC FCD 9126-1 and in Quint2 (see 5[4]), which is also called *extended ISO Model 9126*[Quint2].

3.2 General OS quality aspects

Usually users may experience quality of an operating system different depending on their background. Therefore we distinguish between three groups: *End-users*, *experienced users* and *developers and researchers*. By end-user we understand the broad range of users who use computers for their daily work or entertainment. Experienced users are users with a strong background knowledge of computers and computer science background. Operating system developers and researchers are the last group. Their main interest lies in implementing new features or trying new ways in system architecture. Trying new ways leads to a higher insight in the architecture and design of operating systems: The best way to examine if an idea like a micro-kernel works in practise is to implement one. It is obvious that every of these three groups has a different main focus on an operating systems. As we will see Plan 9 mainly focuses on experienced users, researchers and developers. In the other sections of this paper we will call them simply *users*. Figure 3.2 shows the definitions of quality aspects we think

[4]image taken from `http://carballosa.blogspot.com`

Portability Ability of the software to be transferred from one environment to another

Security Ability to prevent unauthorized access, whether accidental or deliberate, to programs or data

Fault tolerance Ability to maintain a specified level of performance in cases of software faults or infringements of its specified interface

Efficiency (in general) Relationship between the level of performance of the software and the amount of resources used, under stated conditions.

Operability Users' effort for operation and operation control.

Figure 6: [Quint2] quality characteristics relevant for operating systems

are interesting for all operating systems.

A very important feature for non-mainstream operating system is portability. To be able to support legacy (previously written) applications leads to a higher user basis. Having more users for an operating system makes it more likely that 3rd party developers start to write software for it. So the number of new applications appearing for a platform is highly correlated with the number of deployed systems. End-users also tend to reject an operating system if it does not support one or two needed applications [Walfield07]. To quantify portability we could take the amount of available POSIX-software for the operating systems under our scope.

A common interest of all three groups is Security. As the computer worm *Stuxnet* showed in late 2010 security is not only for academic interest - especially if the computers are connected to a network. Many end-users want to do banking business with their computers without the fear that hackers or other unauthorized users steal their data. The same is true for experienced users and researchers and developers. Today security is far more important than in the last century. Later we mention the security aspects of Plan 9, the Hurd and Linux without any concrete numbers because only for Linux reliable quantification has been done.

As the size of software systems is steadily increasing it is more likely that there are errors in these systems. If errors occur in a software an operating system should not crash totally. Software faults may come from third party applications, from errors in the kernel itself or even from hardware failures. Fault tolerance ensures that a system still keeps its functionality and returns to a consistent state even if something unexpected happens. We do not quantify this aspect as the numbers heavily depend on the domain of the systems (supercomputing leads to other figures than home-use of computers).

Suitability Presence and appropriateness of a set of functions for specified tasks

Understandability Users' effort for recognising the logical concept and its applicability.

Analyzability Effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified

Figure 7: [Quint2] quality characteristics relevant for Plan 9

Efficiency is another important quality aspect for operating systems for every group. [Quint2] divides efficiency in the sub-aspects *Time behaviour* and *Resource behaviour*. On the one hand a good time behaviour is essential for interactive applications: End-users expect that something happens if they press a button. If an operating system does not respond fast enough users may be confused. This leads to a bad user experience. In multimedia applications like games it is even more crucial[Walfield07]. On the other hand a good resource behaviour is also important for an operating system: Resources like memory or band-width of a network connection are not infinitely available. If an operating system uses less resources applications running on an operating system have more resources for their calculations. Operating system also need to schedule the available resources in a fair way that every applications gets the resources it needs. As we see later depending on the system architecture this task can be very hard to achieve.

Operability is a sub-aspect of usability. ISO 9126 defines usability as "the capability of the software to be understood, learned, used and liked by the user, when used under specified conditions"[Bevan99]). In practise operating systems are no ends in themselves - they serve to help its user to fulfil a concrete task. So an operating system has to serve the necessary functionality and be as easy as possible in its use at the same time. Also it should be simple to setup the system. A way to quantify the operability is to measure the time a user needs to do concrete tasks. This aspect is crucial for end-users but for researchers of exotic operating systems it is most of the time a subordinate goal.

3.3 Goals and quality aspects of Plan 9

Furthermore, Plan 9 strives for additional quality aspects. In figure 3.3 the definitions of relevant characteristics are listed.

Operating systems need to be suitability. It must offer interfaces to make it possible to use the available hardware and to get the data of the user. End-users need to be able to get their desired functionality either by the operating system directly or with

additional software for this operating systems. For experienced users and developers it also enlarges the suitability of an operating system if they are able to write own small programs or shell-scripts to fulfil their tasks. The trade-off in section 5 shows the benefits of the simple and elegant interface of Plan 9 for this quality aspect.

Understandability is another important usability aspect. If an operating system has easy understandable concepts it is easier for developers to write software for the system. With a consistent architecture and consistent APIs they can reuse the knowledge they gained in one part of a system in other parts of the system. An understandable system makes it a lot easier for experienced users to administrate a system: Tasks like adding users or adding a computer into a network can be done faster. Also end-users profit from an understandable system: They are more productive if they do not need to call the support hotline or consult a manual. As we can see in section 5 the textual interfaces are an advantage for experienced users and developers. Unfortunately the lack of a state-of-the-art desktop interface makes the system hard to understand for end-users. To be fair to Plan 9 we will not focus in the analysis of the trade-offs on this fact, because Plan 9 - at least today - does not focus on end-users.

The last quality aspect we will focus on is analyzability. To be able to debug an operating system it must be possible for developers to analyse what is going wrong, when an error occurs. End-users and experienced users also need to be able to retrieve meaningful pieces of information to be able to make bug reports. On the one hand Plan 9 has strong debugging capabilities even over networks. Due to the fact, that Plan 9 is an hybrid kernel, it is easier than with a monolithic kernel to debug parts of the operating system[Pike91]. But on the other hand we see in the section 5 and 6 the disadvantages of Plan 9 regarding analyzability.

3.4 Sacrificed quality aspects

Although we mentioned that portability is generally an important quality aspect for non-mainstream operating systems Plan 9 does not care about it: Everything is written from scratch. This leads to a cleaner system structure but also leads to a lack of software. APE [5] makes it easier to port POSIX code to Plan 9. But still "[t]here are some aspects of required POSIX behavior that are impossible or very hard to simulate in Plan 9"[Trickey].

3.5 Comparison with Linux

With its two major desktop environments KDE and Gnome Linux offers a far better user experience for end-users. As already mentioned earlier measuring the time a user needs to do concrete tasks is a possibility to get concrete figures for its operability. [Samoladas05] offers figures of a study with average times end-users needed to fulfil tasks like changing a wall-paper, searching a file or writing an e-mail on Windows XP

[5] The ANSI/POSIX Environment

and on KDE 3.1.2. Plan 9 is still far away from a point where such measurements could be done as it still lacks features for end-users.

Furthermore, there exist figures for the reliability of the Linux-Kernel (Linux 2.4.19) in [Samoladas05]. Examinations of a small part of the kernel showed that it has an average of 0.10 defects per 1000 Source Lines of Code (KLSC). The best third in the industry have an average of 0.15 defects per KLSC. Other studies show "that and reliability of the GNU and Linux software was better than that of commercial UNIX products"[Samoladas05].

The Linux kernel itself was designed to be very portable. This was possible due to a very clean design and the concept of loadable kernel modules. Loadable kernel modules made the Linux kernel far more modular and allowed it to split out functionality out of the core of the kernel. With loadable modules it is also not necessary to load functionality which is not needed by a certain computer[Samoladas05].

Linux is used in practise by millions of servers today. Even though intrusions in Linux systems get now and then into public Linux can be called a secure system. Many research has been done on the field of security and weaknesses in the Linux kernel which get public are usually fixed early. Interested readers can find information about security issues of Linux on the Linux Kernel Mailing List[6] or on BugTraq[7].

3.6 Comparison with the Hurd

The system architects of the Hurd wanted to design an operating system which is usable in practise. They noticed that it is necessary to be POSIX-conform to get the benefit of many programs running under their platform. Thus portability was one of the main quality aspects of the Hurd. The developers did not treat POSIX-compatibility as second-class citizens as Plan 9 did. GNU Hurd benefits from this decision. There is a large amount of software running under the Hurd. Even ports of KDE and Gnome exist[8]. But this decision also lead to unnecessarily complicated the system structure [Walfield07].

The Hurd also supports a sophisticated protection and security mechanism. Applications are able to drop capability which they holds. Capabilities under the Hurd give applications privileges like the ability to open ports below 1024 or root-permissions. Giving up capabilities is also called *discretionary authority reduction*. With less authority an application reduces the possible danger an error or a hacker can cause. With the same capability mechanism applications can get more privileges. It only has to prove its identity to the authorization server. Privileges in the Hurd are determined by the identity of the user. But although it is possible for an application to give up privileges users cannot enforce applications to give up privileges they do not need - it still de-

[6] `https://lkml.org/`

[7] `http://www.securityfocus.com/archive/1`

[8] see `http://www.osnews.com/story/10870/GNOME_and_KDE_on_Hurd`

pends on the implementation of the application. But still - if applications make use of this possibility the system in general gains more security[Walfield07].

To be efficient the Hurd is able to circumvents many of the abstractions it introduced. But due to the micro-kernel architecture resource scheduling hard compared to monolithic kernels. It is easier with monolithic kernels to predict the resource demand of an application as more components of the kernel like the I/O-subsystem have data of the scheduler available. A monolithic kernel has a higher-level view than multiple servers in a micro-kernel environment. [Walfield07] describes the problems of resource scheduling of micro-kernels. Plan 9 as a hybrid kernel does not have these efficiency problems.

4 Introduction to trade-offs

We have pointed out before the features of Plan 9. Mostly, we have remarked the changes that Bell Labs has introduced in order to build a new operating system with many advantages. But, as every system, there are some points where decisions have to be made: not every quality aspects can be satisfied simultaneously, and some of them must be forgotten to the detriment of others. In the previous section, we have named those quality aspects that Plan 9 is more concerned with. In this section, we will see some trade-offs, which are the pros and the cons of them, and why we consider two of them the most important, among the others.

The kernel is just a router

The 9P protocol works sending messages between the file servers involved in a request and the router. A user or a file server starts the request and the kernel initiate a chain of messages until the result is got.

On the one hand, letting the kernel be aware of less things is positive for Security, Fault Tolerance and Suitability. If only computation concerned about low-level tasks are made in the kernel, the system will remain alive when an error or an attack occur in a file server. The entire system will not die, just the file server will stop working. For Suitability it is clear that it is more natural to leave the kernel stay in charge only of those things that really has to be, taking out the ones which are not at kernel's scope.

On the other hand, it is negative for efficiency. Just a simple example as the one about asking for the mouse coordinates above can entail multiple messages, see Figure 8. It may make the system very slow, a very important aspect in an Operating System. However, it is not an actual trade-off. Originally, it was, but at the end, the developers of the system thought about it and founded a way of dealing with this problem. This is explained deeply in the next section, how it is implemented and why it is, therefore, not as less reliable as thought.

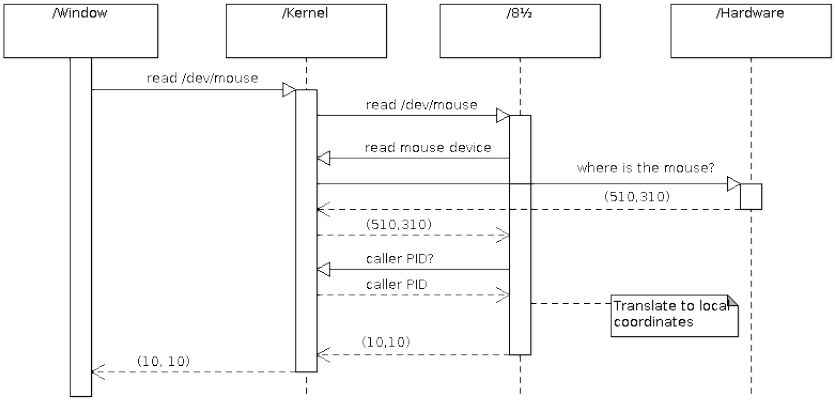


Figure 8: Sequence diagram of a read `/dev/mouse` request

Simple, elegant and human-readable interface

The file-oriented approach provides Plan 9 with a very simple way of communication between the users and the system. The commands are intended to be as human-readable as possible. Just as an example, remind how a remote connection is made: it is only necessary to mount the required remote file system in the local name space, with a very meaningful sentence.

From this point of view, it improves Understandability, Suitability and Operability. But it has also a drawback with the Security, but specially with Analysability. The messages in the system are textual, and they are managed by each file system, they create and use the ones they want and as they want. Therefore, they cannot be easily understood by the programs, they can only be passed to the kernel and sent to the requester back. If a problem has taken place during the transactions it is the user's concern to understand the messages and detect which the problem was and were and why it was generated.

That is also related to Security, as the commands that are written in the files are textual, and again, they cannot be easily understood. But this has also a mechanism for dealing with it, as explained later on in the next section.

Remote errors are transparent

Other of the strong points of Plan 9 is its simplicity when distributing services. Files are treated as local, even they are remote, it is transparent to the user, once some import command has been run. In this case, how can an error in a remote machine can be distinguished from a local one? How can the user know in which machine is the problem? As shown in Figure 3, a (wide) usual Plan 9 installation consists of many fast computers, the *servers*, so, the error could have been generated at any of them and due to any reason.

This third principle of Plan 9 makes the system more Operable, both for users and programmers, in the positive part; but it has to make up for Analysability and Suitability, as the previous named trade-off.

Remote security is transparent/opaque

In relation to remoteness, Security is always a quality aspect to be taken into account. Since the system provides such an easy procedure of remote connection, it is also very simple to loose connection. Concerns about how to deal with security in that aspect must be thought, trying to tip the balance in favor of Operability, minimizing the risks in Security.

4.1 Which are the most important trade-offs?

On the two next sections, these trade-offs are more deeply explained, analyzing how the authors thought about them and which were the decisions they made. One of the most important quality aspects that Plan 9 had to consider was Security. Actually, it

was not until the fourth edition when a file server for Security was included in the release. We will focus on security aspects, and how the 9P protocol works.

But also the remoteness is a very characteristic feature of Plan 9 which creates some trade-offs that have to be considered, so it, and the opacity derived of it, are the others aspects analyzed later.

4.2 Non-trade-offs

The main principle of Plan 9, *everything is a file*, gives many positive features to it, with them not having any drawback. It is also derived from the idea of distributing: services and processes can be isolated by putting them in private local name spaces. As shown before, it has two different advantages: the capability of a user of reproduce his system at any other machine, and have access to all the file servers and functionalities that are distributed over the network, taking advantage of the specialized computation in a fast machine. But it has also another advantage, the focalized location of error: if it takes place in a file server, it stays there, without interrupting other processes running in the system (this is also explained with more detail in the next section, and we will expose why it is on this way).

Likewise, the union folders are also a free advantage of the system approach. It is not necessary the variable `$PATH` anymore: the local directory `/bin` is a union of `$cputype/bin` and `/rc/bin`, the first one contains the binaries and the last the shell scripts, and it could also contain more directories added by the user.

5 Focused trade-off #1: Simple and elegant interface

One of the most important implications of the "everything is a file" idea in Plan 9 is that the system internals are exposed to users (understanding users as both end-users and programmers of the system) in a very uniform way. Programmers do not have to learn about different libraries and system calls, they only need to know how to access a file in a very general way and then which file to access to get the functionality they require.

The key point for showing that uniform interface is the concept of *file server*. A file server is merely a program which exposes its programmer interface through the file system. Those file servers may be *bound* to a place in the file tree, and the kernel takes care of calling the appropiate program each time. The communications is then done by means of the 9P protocol, which will be explained with more detail in the discussion of the next trade-off.

So, in a very high-level view, Plan 9 follows a *broker* architecture: the kernel is only responsible for coordinating communication between processes and file servers, but clients only speaks to the kernel (the broker) which redirects requests to the intented server.

Let's first see an use case where we can spot some of the pros and cons that we may get implementing this idea, and which were balanced during the actual development of Plan 9. We will consider two different scenarios where the user wants to use the mail client that comes bundled with Plan 9, which is shown as a file tree.

The first one is just looking at the mails of the 1st of October. This is done by looking in the `/mail/box/`*user*`/mbox` folder[9]. Each mail is shown as a directory inside this path[10], which several files containing the actual information, such as `subject`, `body`...For our search, the file is `date`. So the problem may be solved by iterating on every directory in the `mbox` folder, and showing those whose `date` file is `1-9-2010`, something which can be easily achieved with a shell script.

This small use case already shows some pros and cons:

- The system is easy to understand, as no special file formats which must be serialized or deserialized enter the game. The user just searches the mail box in a hierarchical way, using well-understood and standard tools such as `locate` or `grep`. So we can see that this kind of interfaces get a good mark on Understandability;

- The fact that we can reuse tools for different tasks makes Plan 9 systems very composable and better-suited for sharing information between file servers (in fact, things like hiding information in propietary formats do not play well with these abstractions). From the point of view of the operating system, this is exactly which we would like to achieve: easy ways to access and share information. So we also get a good mark on the quality aspect of Suitability;

[9] `http://plan9.escet.urjc.es/magic/man2html/1/mail`

[10] `http://plan9.escet.urjc.es/magic/man2html/4/upasfs`

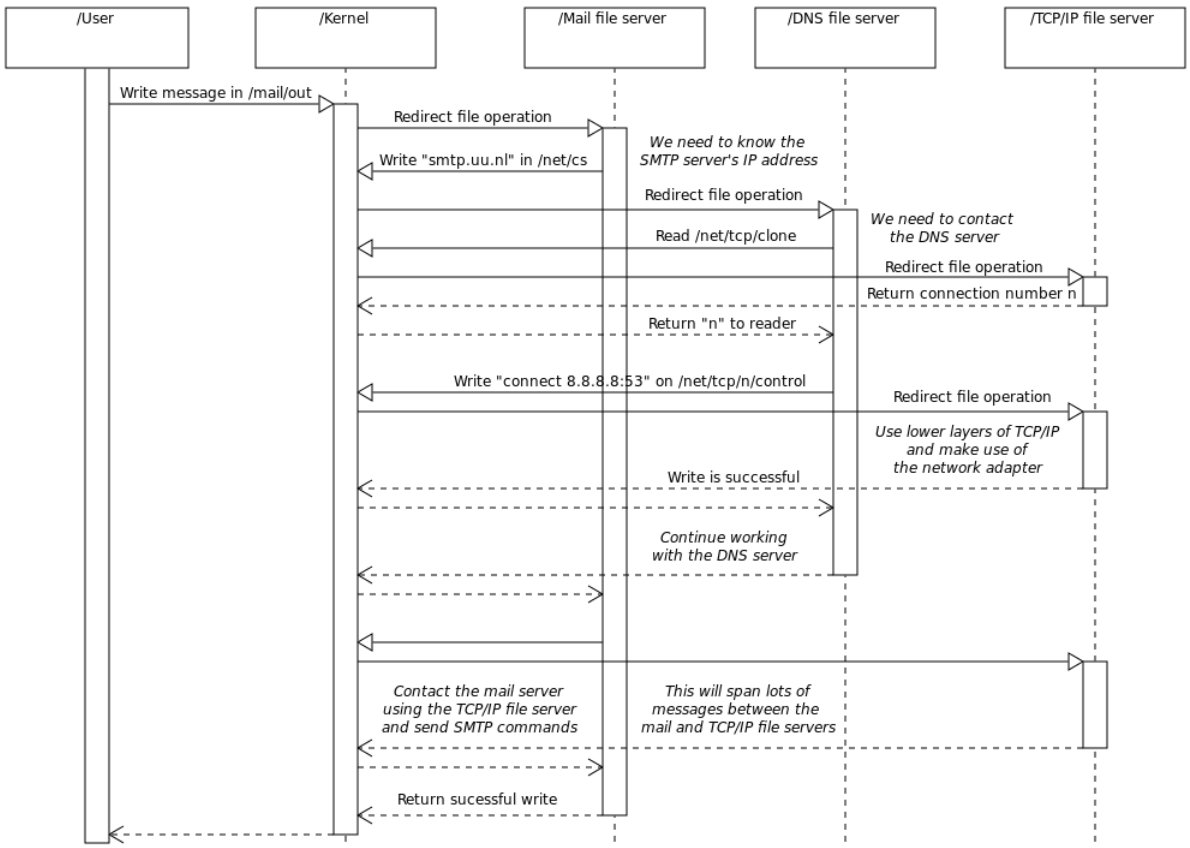
- However, if we don't want to loose any mail from the users that reside in our system, we would need to have the file server listening at every time in behalf of the different users. This brings us the question of integrity and confidentiality of the mailboxes data. The naïve abstraction of "everything is a file" goes much less neat when Security concerns are taken into account: how do we control who can read or write each file? How can we authenticate users?

The other scenario that we can consider is the sending of an e-mail. That may be done in a "Plan 9 style" by writing in a special file in the mail folder. As usual, the information is encoded in a string, not using a binary format, so it may look like

```
to: someone@somewhere.com
subject: example mail

Hi, this is an example mail...
```

The kernel will communicate with the mail file server to make the latter handle the file. The mail server will need to communicate with the network, so it will write in several networking-related files such as `/net/cs` and `/net/tcp`. To get an idea of the actual exchange of messages taking place, we can see the next diagram:



Once again, we can see that we get a very understandable exchange of messages, with well-delimited boundaries for each file server. This also makes easy to replace

file servers in charge of any particular task with newer versions or all-new software, if the exposed set of files is the same, so we get a good mark in Replaceability apart of the pros stated previously.

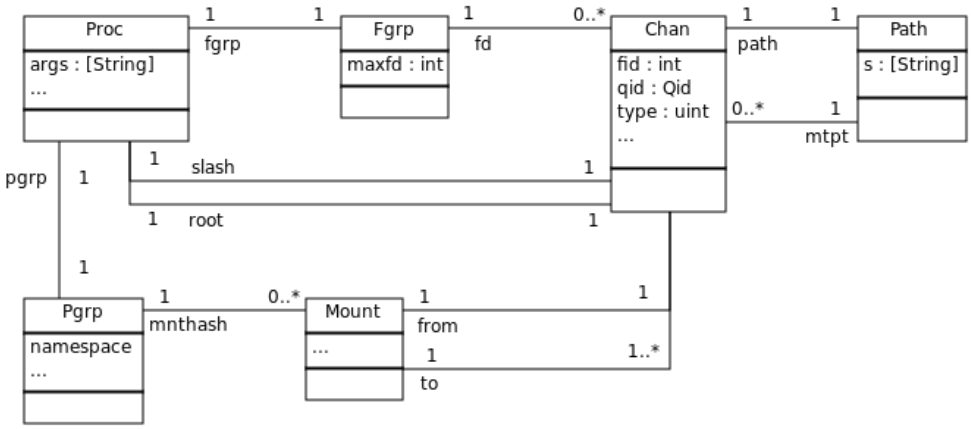
The big con here is Efficiency: we see that messages must all be routed through the kernel, so special care should be taken on it in order to not impose a lot of overhead in this transactions that will be very frequent. We would like to focus on the efficiency and security issues in this section, although more problems are associated with this interface, such as error handling (which will be dealt in the next section) or atomicity of operations.

5.1 Implementation in Plan 9

We would like to outline a bit the implementation of the file access in Plan 9. We will sometimes refer to source files, whose paths should be taken as relative to the source distribution root [Source].

As in most operating systems, in Plan 9 the process is the unit who owns resources. That is, each process has its own set of open files, pipes, locks and so on (whereas, also as in most modern systems, threads are the scheduling unit). There is, however, a difference with the rest of systems: as namespaces are private, each process must own a different mount table and a pointer to what is considered the root / of the file system view.

The developers of Plan 9 grouped the process resources in four groups: process (the name is misleading, but it has the reference to the mount table and gives access to the private namespace), environment, rendez-vous (this is the concurrency primitive in Plan 9) and file (which has a list of open files). The C names given to this concepts are `Pgrp`, `Egrp`, `Rgrp` and `Fgrp`, as can be seen in the `/sys/src/9/port/portdat.h` file. We outline the attributes of these structures, along with some other that will be found in subsequent discussion in the class diagram below. Remark that we show a logical view more than a implementation one: for example, the list of mounts is really implemented as a typical C hash table, so we really find a pointer to the first `from` element, with a list of `to` inside it, and a link to the next mount; but this is difficult to represent in the class diafram and obscures the discussion.



The heart of the implementation is the `Chan` structure, which holds a descriptor to an open file. As Plan 9 is POSIX-compliant, the external interface for file operations (`read`, `write`...) does not take a `Chan` structure, but an integer which is known as file descriptor (also, we would get some stability problems if processes were allowed to play with `Chans` freely). The conversion between them is done with the `fdtopath` function[11].

Now I would like to point to the `read` function[12], specially on this line of code:

```
nnn = nn = devtab[c->type]->read(c, p, n, off);
```

where we are told to read from `Chan` c n bytes beginning in off and save them in p. But as we can see, we do not execute a general `internal_read` function. Instead of that, we go to the `devtab` array, which holds a list of all available file servers in the system, and then select the desired one using the `type` attribute from the `Chan`. So that `read` actually performs whatever reading means in that file server. That is the naïve way of implementing file server operations in Plan 9. We can now make some conclusions:

- We don't have much overhead for file operations. Once a file is opened, the corresponding way of serving the files is cached in the `type` attribute, and with that index we point directly to the file operations. We get no context switch. Note that union folders are not covered in this part, as they have a special treatment. That means that efficiency is no longer a problem, as Plan 9 just uses function pointers to refer to different file servers: they are not as fast as a static function call, but are much faster than a RPC mechanism;

- We see that no additional information is passed to file server. But, as the operations are executed into the same process space, they have access to all information in it. In that way, we can easily implement different views for the same file, as done in the $8\frac{1}{2}$ window manager;

- However, we expected that file servers would run in their own process space, sandboxing them from both the kernel and the processes, and making it easier to share information between different processes using the same file server. However, this is not the case: a bug in a file server, even though cannot make the entire system crash, would crash the entire program accessing to it. So we score less in Reliability.

5.1.1 Problems with paths

In first sight, the only problem attached to the all-is-a-file interface seems to be security. However, we also get a subtler problem, as outlined in [PikeLex]. This problem is that the combination of private namespaces, union folders and different file servers may yield with problems interpreting file names if care is not taken: typical Unix systems implement the current directory as a file descriptor pointing directly to that folder. That makes it easy to traverse a file name relative to the current directory.

[11] Source code in `/sys/src/9/port/sysfile.c`

[12] Ibidem

However, let's consider what may happen if we use that naïve implementation in Plan 9. Let the current directory be `/union`, an union folder. We now change to `./example`, yielding a current path of `/union/example`, where `example` really comes from `/other/path/example`. Now we change the current directory again to `...` What should the new current directory be, `/union` or `/other/path`? If we only save the file descriptor of the current directory, we have no way to desambiguate. This problem is more general and, as shown in Pike's paper, it happens in every system with at symbolic links, so the "bug" is reproducible in any Unix system.

The class diagram for `Chan` shows clearly that this problem is solved in Plan 9 fourth edition by saving the file name used to open the file (this is done within a `Path` structure). For performance reasons, the name is made absolute to the current directory, canonized (that is, with `.` and `..` stripped out) and divided into the different consituents before saving it. Then, `..` is taken lexically: we just strip out the last part of the path and get a new `Chan` which we use to traverse the tree. So, in our example, we would get `/union` back.

Now we can resolve any path as relative to any `Chan`, including the current directoy, and so solving out problem. Even more, that makes easy to have different root folder for each process, because we only need to save which `Chan` is it and resolve paths according to them. In Plan 9 implementation, each process holds `dot` and `slash` file pointers for that matter.

We also have another problem: how do we refer to the real root of the system? If we want to mount a device, how can we refer to it before it is mounted? This is a small trick in Plan 9 implementation, which also blurs a little the hierarchical view of the system. Path starting with `#` are handled specially by the kernel, who knows which is the implementation for them. For example:

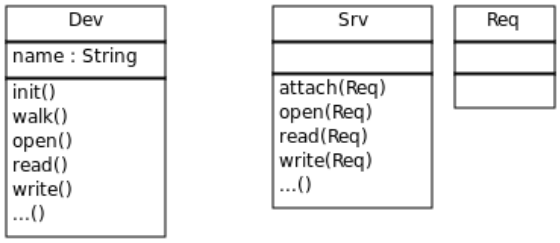
- `#/` refers to the real root of the filesystem,
- `#c` refers to the terminal, as is normally mount in `/dev`, making the `#c/cons` file appear as `/dev/cons`,
- `#p` is the root of the process information file server,
- Drivers for different devices also have special names.

This is a problem for Security, as processes may defeat the sandboxing mechanism of private namespaces using that special names. However, we would not like to impose authentication policies for these low-level file servers, as may make the system very slow. We again see a trade-off, resulting from the actual implementation of the system, between security and performance. This trade-off was not expected (as another solution, such as mounting those special file servers in fixed points in the file structure, may not have so many security issues), but we think that is important to pinpoint them to the reader.

5.1.2 File server implementation and binding

Once file access is clear, we turn to the other side of the coin: how file servers are written and how the mounting process is implemented to be efficient. The associated

structures are shown in the following diagram. We do not show all the attributes and operations for them, only an idea of the relation.



`Dev` is the internal structure saved in the `devtab` array, with pointers to each primitive operation in a file server (whose arguments are not shown in this diagram). Very low level file servers (such as `cons` or `proc`) use this structure directly. The reader may find several examples in the `/sys/src/9/port/` folder, in the C source files starting with `dev`.

The higher-level file servers are implemented using the `9p` library. This library understands the 9P protocol used by Plan 9 and converts the calls into a queue of requests (structures of type `Req`) that are pushed into the file server one by one. Apart from that, some facilities such as authentication are provided for free. The programmer only needs to implement the `Srv` functions, and for most of them already-made ones programmed (for example, if we want to create a read-only file system, we only need to make the `write` call a null pointer, and the library takes care of returning the corresponding error message).

We won't dive more into the protocol, as will be discussed in the next section. But we would like to highlight that `lib9p` implements each file server as a standalone program, and then generates `Dev` structure that communicate with that file server using Plan 9 synchronization primitives such as message queues.

This two different ways to implement a file server is the Plan 9 answer to the trade-off between efficiency (for internal drivers, such as console or graphics adapter, we would write a `Dev`-style implementation, running in-process) and reliability (for higher-level file server, we don't want a bug on it to crash a program using the server): the responsability is passed to the file server programmer.

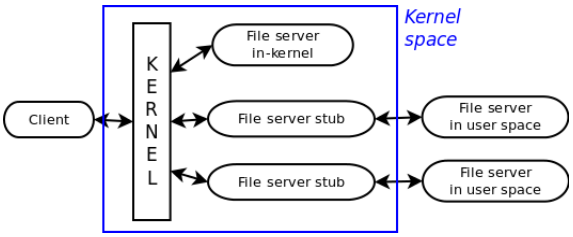


Figure 9: Plan 9 brokered architecture with two kinds of implementation

From the user point of view, mounting and binding are different operations: the first one associates a name with a file server, and the second one is the way of creating union folders. However, for Plan 9 they are the same thing[13]. Each process has a

[13]The algorithm is implemented in `bindmount` in `/sys/src/9/port/sysfile.c`.

mount table as part of the process resource group (`Pgrp`). When we mount or bind a new file, we just add a row to the table, telling that any `Chan` pointing to the `from` file must be redirected to the `to` file. As a `Chan` contains both an unique file number for each file (called the `Quid`) and the file name, we can do efficient matching if the first if known, or use the second to resolve pathnames.

The reader may have noticed that in the class diagram the `to` relation has multiplicity $1 \ldots \infty$. This is the way Plan 9 makes an uniform treatment of mounts and union folders: when a path must be resolved, if we come to a `Chan` with multiple associated `to`s, each one is tried in order before returning an error. This has a trade-off: union are only done at one step, not merging the entire file tree. For example, if we merge folders `a` and `t`, the first containing a file `b/c` and the second a `b/d`, we cannot access to both `b/c` and `b/d` using the union, as `b` will be resolved to the file in only one of the folders, and then continuing searching from that point.

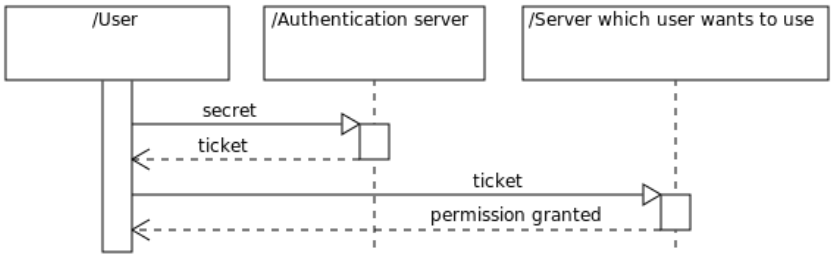
However, we think that Plan 9 developers took that decision because in other way the file name resolution may be very costly (we may need to perform a depth-first search into the tree before returning), whereas the scenario described above is not very usual.

5.2 Security

In the beginning of the section we found that security could be a problem for Plan 9. And until the fourth edition of the system, the only security mechanisms available where typical Unix-like file permissions (although they could be applied to a much larger range of things, as most things are represented as files), sandboxing via private namespaces and flags in the `rfork` call (which performs a similar task to `fork` in POSIX) which allowed to control a limited number of resources (for example, disabling mounts and unmounts in the child) [CoxSlides].

This set of security measures wasn't enough for more complex scenarios, involving a way of authentication other than the one imposed by the 9P protocol or more fine-grained access to resources from file server. For that matter, in the fourth edition on Plan 9 a new, more flexible security architecture, was introduced [Cox02]. Now, the `mount` call used to start a connection to a file server, also includes a parameter `afd` for an *authentication file descriptor*, which should hold the information required for authentication. This authentication must be done previously, by opening the file descriptor which will be used for authentication with the `fauth` system call and then speaking an authentication protocol with normal file operations over that descriptor. In that way Plan 9 is made independent of authentication protocols, as each file server can speak its own.

However, implementing an authentication mechanism for each server, whereas several servers in a machine may share information about users or protocols, is a waste of time. The solution in Plan 9 was the idea of a *authentication server*, a file server which is trusted by the rest. Authentication servers speak a protocol to be sure about an user identity, and then gives back a *ticket* to the process trying to authenticate, which can be user later to log in other file server who trust the authentication one.



All this work of speaking authentication protocols and behaving as an authentication server is responsability of Plan 9 *authentication agent*, called Factotum. In a regular Plan 9 system (one without special file servers requiring extra protocols) this server is the only one which holds *secrets* (Plan 9 way to call passwords). Factotum is implemented as a file server, normally mounted on `/mnt/factotum`, with a small list of files. If we read `ctl` we can see a list of *keys* of the form

```
key proto=p9sk1 user=pepe !password=swarch
```

which introduces a protocol, and additional information used for that protocol (in this case, user and password). Actually, if we read the file the last key-value pair will be represented as `!password?`, as any field marked with `!` is considered a secret and not shown in the screen (even more, the memory region where secrets are kept is prevented by the kernel from being read from other processes or even debugged). Keys are saved in the hard disk by using a file server different from the normal disk file server called Secstore, which uses encryption protocols before writing the data.

With the information in a key, Factotum knows how to authenticate in a remote system. Regular processes make use of that service by means od the `auth` library. Here comes an example (without error handling):

```
fd  = open(file, ORDWR);
afd = fauth(fd, aname);
aip = auth_proxy(afd, amount_getkey, "proto=p9sk1 ..."); // Factotum magic
mount(fd, afd, mntpt, flags, aname);
auth_freeAI(aip);
```

`auth_proxy` handles all the exchange of messages for authentication that previously had to be made by hand. [Cox02] gives more insight on what are the actual messages being exchanged, but we won't enter into that level of detail.

Factotum is implemented as a separate process, that is, using the second procedure for file servers that was discussed in the corresponding section. Processes using the `auth` library only work as a gateway for the exchange of information between two different machines (or the same, if the authentication is local). This opens the gate to single sign-on schemes, where authentication information is shared between different processes, and the user is only asked for the information once (or never, if saved in the Secsctore).

However, Factotum alone is not the cure for all authentication problems in Plan 9. For example, some processes need user changes while running. Typical examples of

that are the `login` process run in boot time, and `cron`, which may run taks impersonated as other user. Permission for the processes to do so are governed by fine-grained capabilities. As usual, the implementation relies on a Cap file server with some difficult message exchange mechanism, although the `auth` library has most of the usual "conversations" already implemented.

Also, file servers must implement their own security schemes, being responsible from denying access to required resources if the user is not allowed to do so. We haven't found information for mechanisms built above file servers and that would allow control lists for file servers.

In conclusion, Plan 9 tries to defeat the loss of understandability and operability of the file system when authentication is taken into account by providing unified architecture and set of protocols for all file servers. If the end user makes use of it, the secrets needed for authentication can be saved and automatically used when necessary, so from an end user point of view all is transparent. This seems to be a good way to solve the problem, as many other authentication agents (Seahorse, GPG Agent) appeared later.

As an addition, Plan 9 developers introduced portability in the authentication protocols: at any point of time, new protocols of authentication can be added to Factotum, and any program using its services has access to that new protocol. This is not an issue that came to our mind when working on this trade-off, but it seems to be a big advantage of the selected implementation.

5.3 Comparison with other systems

5.3.1 With Hurd

As we saw in the overview of the system, Hurd also wants to unify objects in a Unix system, but instead of using files, opts for using intercommunication procedures, specifically *ports*, for giving that common view. Any object in the system is just shown as an object with sends and received messages. Some of the messages are standarised in so-called *interfaces*. One example is the `fs` interface, which is implemented by all processes wanting to give a file sytem-like interface to the outside world.

So Hurd abstraction is, in some sense, more general than Plan 9's, as the file system interface is just a particular case of it. That allows Hurd to move more things to user space: for example, any process implementing the `mem` interface can be used for managing memory in the system (a task usually reserved to the kernel), any process implementing `auth` can be used for authenticating, and so on.

One difference with Plan 9 is that file servers, called *translators* in Hurd, are not bound to files in a per-process fashion, but tied to a specific node in the file tree. Translators can be active (are always running) or passive (its lifecycle is managed by the kernel, and can be started or stopped when it wants). As we saw earlier, in Plan 9 servers are just pointers to some functions, and at programmer's will, those functions can then refer to an outside process using IPC mechanisms.

Although the Hurd approach is more clear and allows better sandboxing of servers, has a very important downside: performance decreases a lot. Whereas Plan 9 has been

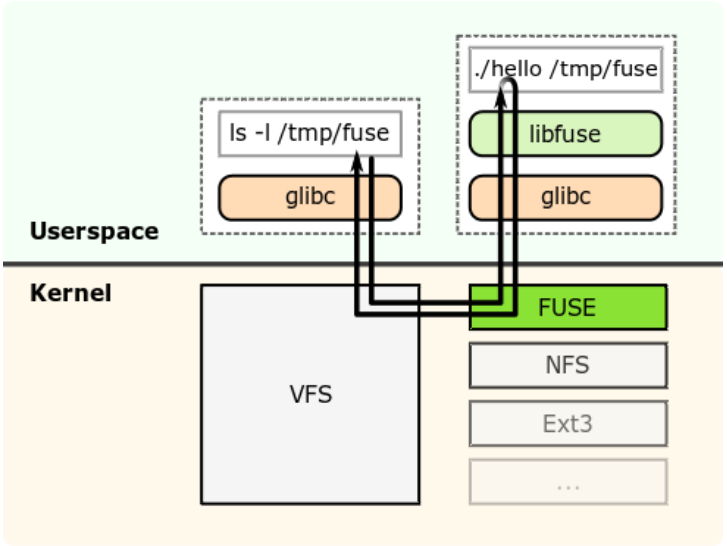
reported to be used as a normal computing platform by several people, Hurd critiques (such as [Walfield07]) always stress the performance part. We may conclude that the point that Plan 9 abstraction has not been taken to the extreme, combined with some clever way to use some file servers without process communication or context switch, has made it usable.

The same critique explains some problems with authentication and authorization when applied to those objects with ports. Hurd security architecture is based on capabilities, but the programmer must explicitly downgrade to the smallest set of capabilities required by the program. Also, as it has no private namespaces, it's difficult to control which part of the file system is shown.

However, we think that Hurd may benefit from some of the implementation decisions made in Plan 9. For example, [Walfield07] cites the `..` behaviour as a problem. But we have seen that Plan 9 has found a way (lexical paths) to circunvent that problem.

5.3.2 With Linux

In the Linux world several we find several projects which try to show information as a set of files in a general way. The first one is a module into the Linux kernel, called FUSE (Filesystem in Userspace) [Fuse]. This module allows to expose a tree of files as normal files in the system. The idea is very similar to the second type of server implementation in Plan 9 (the one involving a queue of requests in a single process). The developer only has to implement a set of methods and FUSE takes care of calling them when needed. Even more, bindings have been created to other languages such as Python or Java.



Even though messages must go inside the kernel and rerouted again to the file server by the FUSE module, the performance of the system is quite good, comparable to in-kernel file systems [Samuel07]. For that reason, newer file system support is now done using FUSE, which gives more stability as the process is in use space. One example of a successful implementation of FUSE interfaces is NTFS-3G, which is the chosen way to read and write into NTFS partitions in many Linux distributions.

In conclusion, Linux has used an architecture for FUSE very similar to Plan 9's. The good results in performance show that this abstraction is indeed useful and enough performant for daily use. We haven't done more research in this topic, but it seems that we can freely mix FUSE with security mechanisms such as SELinux, enforcing better controls and adding capabilities such as Access Control Lists, so most of the security problems would have gone. Of course, what we loose with FUSE is the capability of transparent remote filesystems.

Before FUSE came into being, both Gnome and KDE projects implemented their own filesystem-like abstractions as libraries (known as GIO/GVFS and KIO). With them, new file servers could be added dinamically and easily to the system. However, the use of these libraries was not transparent to client applications: the code had to be rewritten to use them, binding the program to a (sometimes non-portable) library.

In any case, the architecture of those systems is again very similar to FUSE and Plan 9: file operations are routed through the library, which takes care of calling the right file server. Performance is good, because most of the times this routing is done in user space, so no context switch is needed (opposed to FUSE).

6 Focused trade-off #2: Remote errors are transparent

When working with distributed systems, there are a lot of extra environmental factors to take in consideration: connections may be slow, connections may fail, the computer on the other side may shut down or, given a more complex hierarchy than server-client, somewhere deep in the network, some node may be on fire. These are all things that a regular programmer or user doesn't want to be bothered with during simple tasks. This is probably one of the reasons why Bell Labs decided to make remote resources transparently available (i.e. using the same protocol as local resources).

One can even assume that the Pareto principle holds for system architectures: with a rather generic implementation (20% effort), one could facilitate most distributed computing environments in a satisfying manner (80% results). This would mean that, in most cases, anything related to quality of service (QoS) or infrastructure-specific errors can be hidden without causing problems.

Unfortunately, there's still 20% unaccounted for. What if the nature of the network errors is relevant? What if one needs control over bandwidth usage or network timeout times? Since we've hidden these concepts inside the responsible file servers, remoteness has become a black box.

In general, there are three quality aspects involved in this tradeoff. Hiding all these errors and QoS concerns may make the system *more Operable*, since the user is not bothered with details during simple tasks, but makes it *less Analysable* and *less Suitable*, since the errors are likely harder to analyse and troubleshoot for both user and program. To see how Plan 9 deals with this problem, we actually split it into three parts: automated error handling, manual error handling and control over quality of service.

6.1 Automated error handling

In Plan 9, any system call may fail arbitrarily (because file systems may say when a file operation has failed). When this happens, the system call that failed returns a sentinel value (-1), to which programs should check the value emitted by the `errstr` function. The 9P protocol exactly copies the most important system calls and has a similar way to communicate errors. The error message is supposed to be a human-readable text that describes the error. As stated by the 9P manual, these texts are "only advisory and in some sense arbitrary". The most important implication is that there are no general machine-readable semantics attached to the error messages. What's worse, the file servers may truncate error messages as they see fit. This makes these error texts seem unreliable carriers for descriptions of errors.

Consider for example a file server that talks to a remote HTTP server. It is possible to convert (a subset of[14]) the methods in the HTTP protocol to the standard file operations in Plan 9. This works fine if the methods don't fail, but if any HTTP response has a status code outside the 1xx or 2xx range (continuations and successes), it is not possible for programs to distinguish between a temporary error (e.g. server is too busy), a permanent error (e.g. the file doesn't exist), a request for authentication (which can be

[14]One would have to choose between POST and PUT, but seeing the use of these commands in on the current Internet, PUT can be sacrificed without problems.

resolved by retrying the request with the proper credentials) or a redirection (which could be resolved automatically) when the program can't parse the HTTP status codes. Especially redirections would be difficult to handle using the existing error system in Plan 9. An option would be to let the file server take care of it, but then there would be a loss of control over redirection policies (e.g. what domains or paths are allowed to be followed, or how many recursive redirects are allowed).

Upon inspection of the source code, it seems that no program is interested in the nature of the error. An example taken from `/n/sources/plan9/sys/src/games/life.c` illustrates this:

```
if ((bp = Bopen(filename, OREAD)) == nil) {
    snprint(name, sizeof name, "/sys/games/lib/life/%s", filename);
    if ((bp = Bopen(name, OREAD)) == nil)
        sysfatal("can't read %s: %r", name);
}
```

Most applications show similar patterns: either try to work around problematic operations by just trying something else, or simply halt the program with an error message. This gives the impression that the authors of Plan 9 never intended to have sophisticated error handling. This doesn't mean that it's impossible to have it anyway; one can work around this problem in some ways.

- Use logging, either with the local `srv` server or on the remote file server, and inspect these logs when errors occur. The error messages are just summaries of the log entries.
- Let `srv` add another hierarchy with metadata about the connection to the remote server. This metadata can also be read when importing namespaces recursively.

6.2 Manual error handling

One can imagine that, since all errors are communicated in textual form, a file server like `srv` can give specific errors when something went wrong in the transport layer, clearly distinguishing errors like "connection interrupted" and "remote machine on fire" from "file does not exist" and "access denied". While programs will still have a hard time to parse these error messages, a human user should immediately be able to see why his program isn't working. In fact, these strings could contain even more useful information; in the case that one would recursively mount remote directories, one could localize the point of failure with less effort.

For instance, imagine that Alice, Bob and Carol are all using a Plan 9-like environment. Alice mounts a file from Bob's machine that Bob mounted from Carol's machine, and Alice tries to read this file. While reading, the connection between Bob's and Carol's machine fails. With error strings, it would be possible to have an error message like "connection to machine 'carol' failed" instead of just "connection failed" or "server error", making it much easier for Alice to analyse and troubleshoot her problem.

Unfortunately, we haven't checked whether the implementors of Plan 9 have exploited this opportunity. However, seeing that many programs simply prepend their

own error message, it is likely that a user ends up recieving a causative chain of error messages (somewhat like an exception trace), which still gives some more information than the "topmost" error. These chains may still become truncated, so there is a slight danger of information loss involved when no logs are kept of the activities of a file server.

6.3 Quality of service

Quality of service is a term that describes reliability and availability of services. In the context of distributed file systems, this mostly includes bandwidth management, priorization, latency calculations, timeouts and policies for resending messages that weren't answered. All these considerations are not handled by 9P itself, but by its transport layer.

As of the fourth release of Plan 9, TCP/IP is the de facto transport protocol to be used with 9P. Previously, Plan 9 used its own protocol called IL, but that protocol proved to be useless for long-distance connections. TCP/IP does take care of most quality of service aspects. The design of TCP/IP has some downsides for certain situations: high-latency connections often have less bandwidth available [Gu07], and since unreliable or out-of-order transport is not supported (while it could prove useful for 9P, since it doesn't need to be in-order), packet loss can also drastically cut into the performance of such connections.

Besides this, Plan 9 does not seem to give control over the QoS parameters of its TCP/IP stack. The `srv` command doesn't even take a timeout parameter like many other file servers do.

An example where control over quality of service would be important is in an environment where the system's administrator needs to have full control over the bandwidth usage priorization. This could be a server that hosts different kinds of data. For video streams, it is much more important that the data arrives soon than that all data arrives (i.e. a little packet loss is not a problem). This also holds for video game protocols, although some protocols are heterogenous in this respect (some parts of the protocol require reliable, in-order transfer, such as chat messages or scores, while other parts are a bit more lenient, like object movement).

Since the control over QoS parameters in Plan 9 seems limited or non-existant, there are a few ways to work around this:

- Keep the process on which a QoS policy must be imposed in a 'jail', which transparently takes care of timing and balancing by wrapping all system calls.
- Use an external firewall or router to reshape traffic. There are routers that prioritize or even fragment TCP/IP packets in order to achieve good QoS.
- Use a different transport layer that is more suitable than 9P. This would be a very suitable option, since there are transport layers available that have more sophisticated QoS mechanisms than TCP/IP has.

6.4 Comparison with other systems

6.4.1 Linux

Probably the most important 9P-like protocol for Linux and UNIX distributions is NFS (Network File System). We will consider the most recent NFS version at the time of writing, namely NFSv4 (as defined by RFC 3530).

Inside the NFS protocol, errors are communicated as numbers with fixed semantics. This means automated handling of errors should be possible with less effort than in Plan 9: even if a program has to guess the exact parameter or condition that was the root cause of an error, NFS gives a machine-readable hint, while in 9P, error messages must be parsed or partially compared to get similar knowledge about an error – if the parsing succeeds at all. In this respect, NFS (or more broadly, POSIX-like file interfaces) might be a more suitable protocol than 9P for programs that need to analyse errors.

However, error numbers are the only response sent when a request fails. There is no first-class way of communicating "custom" error messages. This means that there is no way to give more precise errors than the pre-defined errors, which have fixed semantics. Unlike in Plan 9, where any error classification can be implemented, this could make errors less analysable.

6.4.2 Hurd

We recall to the reader that the basic form of communication within Hurd is based on ports. The Mach microkernel ensures that messages sent over this medium are received reliably and in-order, so the programmer doesn't have to care about that. This can be done because Hurd translators and servers are not designed to operate in a networked environment, only in a local one (which is different from Plan 9's approach). In any case, this kind of support could be added because Hurd systems calls are allowed to return an error on allocation and use [Valderrama], and several error codes could be used, following the familiar Unix semantics.

The messages exchanged by ports can be completely arbitrary, and each protocol is expected to implement its own error mechanism. However, if we are using the Mach Interface Generator (MIG), we can just forget about the errors and describe the actual interface of our protocol in a easy way. For example, this is the interface description for the `fstat` call in a file system:

```
routine file_statfs (
    file: file_t;
    RPT
    out info: fsys_statfsbuf_t);
```

But each of this routines also return a code telling about success or failure of the system call. We can see an use of it in the documentation for `dir_link`: "If `dir` and `file` are not implemented by the same filesystem, `EXDEV` should be returned." [DirLinkDoc].

In conclusion, Hurd error handling is done in a primitive way, by using error codes. This is very efficient, but we can lose some of the information that we would keep if exceptions were used, for example. In any case, we think that Plan 9's way of signaling

errors, by means of a global string, again has more downsides regarding suitability (for automated error handling) than the way it is done with Mach, especially since it leaves room for custom error schemes.

Regarding QoS, we're led to believe that the Hurd project is still lacking. Marcus Brinkmann, one of the more active developers of GNU/Hurd, has recently stated that QoS is in fact an open challenge for systems in general, which would imply that this holds for Hurd as well [HurdOpenIssues]. There has been research to implement quality of service control in some variant of the Mach kernel [Kawachiya95], but we don't know anything about the current state of this aspect of Mach.

7 Conclusions

7.1 Summary

The Plan 9 operating system focuses on the following principles:

- 1. Everything is a file
- 1+ Most names and contents are human-readable.
- 2. Heavy use of namespaces
- 3. Transparent file systems

Legacy support with a POSIX compatibility layer is only threaded as a second-class citizen. Instead the developers focus on creating their own pure environment. The Plan 9 operating system can be used in practise by advanced computer users for research or running a simple web-server. Plan 9 also shows how its simple principles lead to an elegant system architecture. But the price for this elegance are trade-offs in the implementation: The developers had to solve problems with efficiency and ambiguous paths to get its simple and elegant interface.
For end-users Plan 9 still is not usable: It is very inconvenient in its use and it does not offer important applications for daily work like web-browsers or word-processors.

7.2 Influences of Plan 9

The developers of Plan 9 also invented the UTF-8 encoding of Unicode. Unicode is an international standard for characters in binary code. Plan 9 was the first operating system with UTF-8 support [Raymond03]. Today UTF-8 is the most important character encoding in the internet. Unicode support in operating systems is important for the international use of computers even if different non-ASCII characters are in use.

Plan 9 also influenced the design of other operating systems. Today Linux offers the virtual file-system *procfs* as an interface to various configuration parameters: Users can view or alter system information in the folder `/proc`. With FUSE it is possible today to implement file-systems in user-space. The Wikipedia-article `http://en.wikipedia.org/wiki/Filesystem_in_Userspace` shows many examples of things possible with FUSE[15]. The popular Unix desktop environments GNOME and KDE developed for transparent remote file access GVFS[16] and KIO[17] respectively. Unionfs[18] makes it possible to unify several directories in Linux in a similar way as Plan 9.

7.3 Future of Plan 9

In 1995, a successor to Plan 9 called Inferno started its development in Bell Labs. Inferno has the same principles as Plan 9 but many parts have been reimplemented in

[15] e.g. with FTPFS the user can mount a ftp-directory directly into his file-system
[16] see `http://fedoraproject.org/wiki/Features/Gvfs`
[17] see `http://everydaylht.com/howtos/desktop/kio-slavery/`
[18] see `http://unionfs.filesystems.org/`

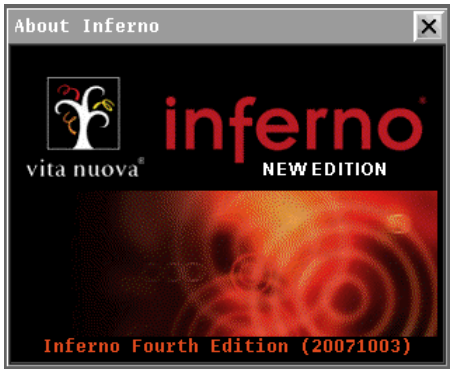


Figure 10: About screen of Inferno 4th edition

the programming language Limbo which was especially designed for Inferno. Limbo generates portable bytecode which can be interpreted by a virtual machine. The Inferno kernel offers a virtual machine. Thus Inferno is extremely portable across several architectures[Dorward97]. Inferno was sold to Vita Nuova Holdings in 2000 and has finally been open sourced in 2004. Plan 9 and Inferno are very close related: Some of the latest additions of Plan 9 were first introduced in Inferno. The latest version of the 9P protocol, which is a fundamental part of Plan 9, is identical to Styx, the protocol of Inferno. Due to its implementation on Limbo there exist native ports of Inferno on the most common processor architectures (e.g. ARM, x86, PowerPC). It is even possible to run Inferno on a virtual machine under host operating systems like Windows or Linux. Thus Inferno is a complete virtual operating system. Inferno has commercially been used in firmware for IP switches and routers, but there is no widespread use in other areas.

7.4 Further research

Although it may look different on the first sight there is still activity in the research of Plan 9. In the last five years the researchers of Plan 9 held every year an international workshop for interested developers and students. Recently Bell Labs ported Plan 9 to different architectures like ARM. In 2010 the researchers even ported Plan 9 to supercomputers and started to work on a Linux emulation[Minnich10]. Linux emulation is important for Plan 9 to get more software running as it is not fully POSIX-compatible.

For further research on the Plan 9 architecture other features and different aspects of the operating system could be examined. Details of tools provided by Plan 9 like its programming language Alef or the general-purpose editor acme may be very informative.

8 Questions & Answers

Group A

Mark Rouhof
"Different computers would handle different tasks: small, cheap machines in people's offices would serve as terminals providing access to large, central, shared resources such as computing servers and file servers.". The authors gave a structure of a large Plan 9 installation (Figure 1). But what are the minimal amount of resources you need to install Plan 9?

Any recent computer is enough for running Plan 9. Of course, if you want to use remote file servers, you need some kind of network connectivity, although in principle it is not required.

Kevin Ingen
Modern software programmers tend to use high-level programming languages, mostly due to some kind of cost-benefit analysis done by some manager. Is there a way to incorporate high-level software like a Java Enterprise server on a Plan 9 server?

Of course, you can use any Java software if you port the JVM to Plan 9. We don't know how difficult it would be. There is a POSIX compataibility layer, but the system as a whole is not entirely POSIX, so maybe you would run into some problems.

Sjors Otten
Plan 9 is very flexible and has an understandable protocol and philosophy (architecture) behind it. I like the fact that everything is seen as a file and that the kernel is nothing more than a router. My question however concerns the idea behind the filesystem. Everything in Plan9 is seen as a file (also keyboard, mouse, screen) allowing it to mount with `mount --vfat --t /dev/mouse` or something like that. What happens when this is done, and how does it affect the system/terminal/namespace? How is the tradeoff between maintainability and usability by end-users? I can imagine that the maintainability is good (due to Plan 9 only having 50 system calls or so) but how does it relate to and affects usability for end-users?

For the first thing, trying to mount the mouse would give an error, as that file may not have the required headers for a VFAT partition.

For the trade-off between maintainability and usability, we do not see any problem. Plan 9 tries to be easier giving an homogeneous view, so developers would write easier to understand, and then more maintainable; and users would only see files, so the system should be more usable for them. In any case, users would interface with desktops like Gnome or KDE, that would hide complexity for them.

Robert Vroon
Plan 9 seems to be a transparent system. Everything is a file and network files are as easy to access as local files. But what about the security of Plan 9? For example, the security of confidential files.

Look at the section about Security in Trade-off #1.

Group B

Alessandro Vermeulen
1. What do the authors of Plan 9 plan to achieve with their implementation of concurrency? It seems to me that it only has drawbacks? 2. How does plan9 account for overhead? The way I understand it, Plan 9 has a lot of overhead because of all the indirection. I know this is useful for Security and stuff, but isn't there a way to apply fusion of these systems? Fusion in the sense that multiple programs are dynamically fused, or composed, to expose one interface to the outside, while maintaining the flexibility of separate programs and in the same time reduce the communication overhead.

1. We think that the primary purpose was to build their kernel without caring about locking and low-level concurrency primitives. Maybe their library is not well-suited for every single concurrency problem, but it is enough for the communication done inside the kernel. This is something that is done in almost any other modern programming language or platform, adding some abstraction (actors, channels...).

2. As you can see in the File server implementation part on Trade-off #1, you can implement file server in a way that does not involve any process communication, so you don't lose that much efficiency. There are places where fusion could be improved, like sharing buffers between file servers and clients using that data, but Plan 9 does not seem to implement any of these.

Lambert Veerman
After reading the paper on Plan 9 you really believe it has a solid architecture. All architectural decisions seems to be well considered. Still I had never heard of the system before and (correct me if I'm wrong) it seems not to be used by many people, although it is not only a research project but ready for daily use (they say). What are the major drawbacks of using the system in practice, there must be some architectural decisions that made the system impractical for daily use. (Or is it just a case of bad marketing?)

We mostly agree with [Raymond03]: Plan 9 doesn't give many advantages to end-users compared to the rest of operating systems, while lacking lots of applications that cannot be trivially ported to Plan 9. So, for end-users, Plan 9 is not really ready for consumption. In any case, their primary group was researchers and advanced users.

Kevin van Blokland
The Plan 9 O.S. sounds really interesting. Defining a more abstract way to handle system requests, API calls etcin terms of fileservers. One advantage of this operating system is the ability to define views on the filesystem. In this way programs even have a more abstract way to communicate with other components. This functionality is implemented by using namespaces. (Am I right?) It sounds to me that different process and user applications all run in there own namespace, which sometimes share an overlap with another process / applications. It is the namespace implementation that also ensures certain entities do not access certain resources (security). In essence the system needs to maintain these namespaces. It was not clear to how these namespaces where actually implemented. It also seems to me that is

maintaining these namespaces can be very complex in terms of operations required. How does Plan 9 implement the namespaces, and do the namespaces cause a lot of overhead?

Keeping it short and extending a bit what it is said in the tradeoff #1 section, each process has two "local" variables that refer to it. First of all, they have a `root` field that points to the `Chan` that must be considered as `/`. In that way the process only can look to information under that folder, and if you create a blank one and mount that only what you want to be accessible, you get a private sandboxed namespace. Apart from that, each process also gets a mount table (well, actually a mount-and-bind table), so file server connections can also be made private for a process.

Normally, these structures are inherited from the parent process when creating a new child, but they can be controlled with special flags for the `rfork` system call (Plan 9's replacement of `fork`). In any case, implementing them only causes a small memory overhead, because we have to save the information for every process. But the access to the tables is the same as in any Unix system (only the kernel chooses the table), so no time performance penalty is incurred.

Thijs Alkemade
Plan 9 was designed in a time when computers were huge and the idea of moving one while it was on was ridiculous. How well is it adapted to the use on laptops or hand-held devices where the network might change any moment?

We think that it is still relevant: with the advent of Internet, more and more information is held on external servers. Plan 9 could be easily adapted to work in cloud environment, replacing by file servers things like Dropbox. Plan 9 is very well-suited for network: they tried to use Plan 9 in high-performance computing and their results were good.

Group C

Matthias Lossek
As accessing files for everything sounds a little bit slow, I want to know: Are there any performance tests comparing UNIX and Plan 9, and if not not, would it make sense comparing the two different approaches?

Unfortunately, there is no direct comparison. But, as told in the tradeoff #1 section, the Linux kernel plus FUSE shows a similar architecture to Plan 9, and they are reported to work very well, so we expect Plan 9 to do so.

Tim de Boer
Because everything in Plan 9 is working together in a network, it's important the "main" servers (like the file server) keep working correct. What I missed in the article is; how does Plan 9 deal with crashes of these vital chains in the network? What happens with the terminal stations and how do they recover from this?

See section about Quality of Service in Tradeoff #2.

Rik Janssen
After reading the article the physical architecture on which Plan 9 is built seems to be important to for fill specific software features of plan 9. An example that is stated in an article about Plan 9 issues the requirement of security and the how this is influenced by the hardware: "Plan 9 does not address security issues directly, but some of its aspects are relevant to the topic. Breaking the file server away from the CPU server enhances the possibilities for security. As the file server is aseparate machine that can only be accessed over the network by the standard protocol, and therefore canonly serve files, it cannot run programs.". The security requirement is met because the file server and CPU are physical separated from each other, could you come up with another change in the physical architecture that will make it harder to for fill a particular software quality aspect that is important for Plan 9?

As we saw in section 2 another important quality aspect is *Analyzability*. With Plan 9 it is easier to debug and analyse bugs in software running on a remote machine. This can be especially useful if you split your development environment into a machine where you run the software and a machine where you develop and analyse/debug the software. They even can use different processor architectures with Plan 9.

Group D

Renato Hijlaard
Is there any 3D support in Plan9? If everything is done through files, will it not be to slow to run any 3D graphics?

Plan 9 was not designed with 3D in mind. In any case, efficient implementations could be done by using an in-kernel file server.

Hans Peersman
Is it possible to connect Plan 9 to another OS, for example Windows or OS X? Can they "talk" to each other?

There are two ways to "talking" of sharing files:

- Windows or Mac OS X could implement an 9P layer, so file servers could be shared transparently. There is Plan 9 from Userspace project which allows to implemente those file servers in UNIX, and also a FUSE module for 9P in Linux;
- Plan 9 could implement SMB or NFS as a file server. Indeed, it does so for FTP transactions, which are exposed in the file server.

Nikos Mytilinos
In the paper you have suggested we can read that with Plan 9 "the same operating system runs on all hardware. Except for performance, the appearance of the system on, say, an SGI workstation is the same as on a laptop." We know that there is such a strong bond between hardware and software; every manufacturer, since it is the

only one who knows best how its hardware works, develops drivers to support various operating systems and, thus, guarantees the performance of its systems. Even software houses are producing different versions of their programs in order be benefitted by specific hardware configurations. Hence, I would like to know if Plan 9 allows to support drivers, for instance, to increase its performance on configurations that include advance hardware. Does this approach to neglect latest developments in hardware and focus on software ubiquity put a threat in Plan 9's overall performance?

As in system, drivers have to be written for the different devices. The idea in Plan 9 is that all the drivers sharing common patterns share a common interface as a file server, there is no reason to be slower than in other systems.

In any case, you can still write drivers for specific devices, or communicate by binary data instead of human-readable strings (the latest is only a recommendation). But on the other hand, if you don't focus on one specific device, you can write more portable applications.

Jeroen van der Velden
Can the IL protocol also communicate with computers using other internet transport protocols?

Yes, you can. IL runs over IP like UDP or TCP does. If it is not possible to use IL directly in an environment because packet filtering is used it is still possible to tunnel a connection with TCP/IP. But as section 6 shows today IL is mostly abandoned and TCP/IP is used directly.

Group E

Sander van der Rijnst
Do you consider the Plan 9 project as a failure? Could you describe that from a commercially point of view and from a research point of view?

It's obvious that Plan 9 hasn't been the chosen platform around the globe, but we think that from a research point of view, Plan 9 is a big success: its direct descendant Inferno is used commercially and lots of other projects are using their ideas (see section on Influences in the Conclusion).

Geert Wirken
Is it easy to port traditional Unix applications to Plan 9? Are there tools or libraries that assist in adapting applications to the Plan 9 environment?

Refer to [Trickey].

Theodoros Polychniatis
If you try to download plan 9, you will see that the last release is June 2003! What does this mean? Is the project active? If yes why the last release was 7 years ago and if no why the development stopped? What are the main problems that keep the

project evolving so slowly?

Nowadays the development is done with a daily builds basis, so there is no official latest version release, but development is still happening. Research is still going on: there have been several conferences in the last five years.

Ruben van Vliet
What are the advantages/disadvantages about presenting everything as a file? Is there any 3D support?

For the first question, just read the section about trade-offs. For the 3D question, see the answer to Renato Hijlaard's question, group D.

Group G

Jacek Marek
What is the reason not to use super-user account and have other administrative accounts instead? What is more, isn't the use of `none` account unsafe?

It is still possible to have a de-facto root account. You can do it if you do not give up any rights in the boot process. But having a root with all possibilities is not a good idea. To have an almighty root account makes it easier to intrude into a system. In Linux there are several efforts to get rid of it when secure environments are required. See SELinux at `http://www.nsa.gov/research/selinux/index.shtml` or AppArmour at `http://www.novell.com/linux/security/apparmor/` for more motivations why you should do this.

Vladimir Smatanik
What is the difference in behaviour of `mount`, `bind` and `unmount` system calls regarding union directories in comparison to other popular UNIX systems (e.g., RHEL or SLES)?

First of all, in normal Unix systems, union directories are not usually part of the kernel, and they are implemented separately, so mounts for that are usually done in a different way from normal mounts.

In any case, the mount concept in Plan 9 is very different from Unix system. In Plan 9, it just adds a new row in the process mount table, with a reference to the root file for that mount. In Unix, kernel structures are involved, as the mount information is system-wide. Also, the name bind is usually used to refer to mounting an already mounted filesystem in another place, whereas in Plan 9 it usually refers to adding new files to a union folder.

Richard Derer
Is there any way to use Plan 9 without having access to a mouse?

You don't need a mouse to use Plan 9. However, rio (the graphical server) need a three-mouse button to be used, and most of its functionality can only be accessed that way.

Alexandru Dimitriu
How does plumbing work in Plan 9?

The best source of information is [PikePlumbing]. But, in short, plumbing associates a regular expression with an action to be performed. When plumbing is requested to work, it looks for some expression that matches the selected text, and if matches, executes the action. This is heavily used in Plan 9 window manager to provide some kind of better links.

References

[Ballesteros07] Francisco J. Ballesteros. *Introduction to Operating Systems Abstraction using Plan 9*. Universidad Rey Juan Carlos de Madrid, 2007.

[Bevan99] Nigel Bevan. *Quality in use: Meeting user needs for quality*. Appeared in *The Journal of Systems and Software 4*.

[BrikmannHurd] Marcus Brinkmann. *The Hurd, a presentation*. A talk about Hurd for OSDEM, Brussels, 4. Feb 2001, Frühjahrsfachgespräche, Cologne, 2. Mar 2001 and Libre Software Meeting, Bordeaux, 4. Jul 2001.

[Collyer10] Geoff Collyer. *Recent Plan 9 Work at Bell Labs*. `http://iwp9.quanstro.net/slides/9ports.pdf`.

[Cox02] Russ Cox et al. *Security in Plan 9*. Appeared in *Proceedings of the 2002 Usenix Security Symposium*, San Francisco.

[CoxSlides] Russ Cox. *The Plan 9 Security Architecture*. Slides available in *swtch.com/ rsc/talks/sec-spain.ppt*.

[DirLinkDoc] Documentation on dir_link routine, available from `http://www.gnu.org/software/hurd/hurd/interface/fs/23.html`.

[Dorward97] Sean Dorward et al. *The Inferno Operating System*. Originally appeared in the *Bell Labs Technical Journal, Vol. 2, No. 1, Winter 1997, pp. 5-18*.

[Fuse] Fuse homepage. `http://fuse.sourceforge.net`.

[Gu07] Yunhong Gu and Robert L. Grossman, *UDT: UDP-based Data Transfer for High-Speed Wide Area Networks*. Appeared in *Computer Networks (Elsevier). Volume 51, Issue 7. May 2007.*

[HurdOpenIssues] Open issues in Hurd regarding multiprocessing, available from *http://www.gnu.org/software/hurd/open_issues/multiprocessing.html*.

[ISO08] *Selection and use of the ISO 9000 family of standards*. `http://www.iso.org/iso/iso_9000_selection_and_use.htm`.

[Kawachiya95] Kiyokuni Kawachiya and Masanobu Ogata and Nobuhiko Nishio and Hideyuki Tokuda, *Evaluation of QOS-Control Servers on Real-Time Mach*. Appeared in *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video, 1995, pp. 123–126*.

[Minnich10] Ron Minnich. *Linux emulation*. `http://iwp9.quanstro.net/slides/linuxemu.pdf`.

[Overview] Plan 9 Overview. `http://plan9.bell-labs.com/plan9/about.html`

[Pike91] Rob Pike et al. *Plan 9 from Bell Labs*. Appeared in *Computing Systems 8 #3, Summer 1995, pp. 221-254*.

[PikeFourth] Rob Pike. *Changes to the Programming Environment in the Fourth Release of Plan 9* http://doc.cat-v.org/plan_9/4th_edition/papers/prog4

[PikeLex] Rob Pike. *Lexical File Names in Plan 9 or Getting Dot Dot Right*.

[PikeNames] Rob Pike et al. *The Use of Name Spaces in Plan 9*. Appeared in *Operating Systems Review, Vol. 27, #2, April 1993, pp. 72-76*.

[PikePlumbing] Pike, R. *Plumbing and Other Utilities*. http://doc.cat-v.org/plan_9/4th_edition/papers/plumb

[Plan9FAQ] Plan 9 Wiki. FAQ. http://www.plan9.bell-labs.com/wiki/plan9/FAQ/index.html

[Praxiom10] Praxiom Research Group LIMITED *ISO 9000, 9001, and 9004 Quality Management Definitions* http://www.praxiom.com/iso-definition.htm.

[Quint2] *QUINT2: the Extended ISO Model of Software Quality*. http://www.serc.nl/quint-book/.

[Raymond03] Eric S. Raymond. *The Art of Unix Programming, Revision 1.0, Chapter 20.* http://www.faqs.org/docs/artu/plan9.html

[Samoladas05] Ioannis Samoladas. *Assessing Free/Open Source Software Quality*. http://opensource.mit.edu/papers/samoladasstamelos.pdf.

[Samuel07] Samuel, C. *Comparing NTFS-3G to ZFS-FUSE for FUSE Performance*. Blog post available in http://www.csamuel.org/2007/04/25/comparing-ntfs-3g-to-zfs-fuse-for-fuse-performance

[Source] Plan 9 source code. http://plan9.bell-labs.com/sources/plan9/

[Trickey] Howard Trickey. *APE — The ANSI/POSIX Environment*. http://doc.cat-v.org/plan_9/4th_edition/papers/ape.

[Valderrama] Manuel Pavón Valderrama. *The Unofficial GNU Mach IPC beginner's guide*, section *A Little IPC Project*. Available from http://www.nongnu.org/hurdextras/ipc_guide/mach_ipc_basic_concepts.html

[Walfield07] Walfield, N. and Brinkmann, M. *A Critique of the GNU Hurd Multi-Server Operating System*.

Appendix: Distribution of the work

- Section 1. Introduction: Johannes
- Section 2. Overview: María
- Section 3. Quality aspects: Johannes
- Section 4. Trade-offs: María
- Section 5. Trade-off #1: Alejandro
- Section 6. Trade-off #2: Stijn
- Section 7. Conclussions: Johannes
- Section 8. Questions: all the members of Team F