Polytechnic University of Cartagena

School of Telecommunication Engineering

DISTRIBUTED SYSTEMS AND SERVICES

PRACTICE 6: NFS NETWORK FILE SYSTEM

Teachers:

Antonio Guillén Pérez
Esteban Egea López

Index

1. OBJECTIVES 3

2. INTRODUCTION 3

3. FILE SYSTEMS 3

3.1. Deploy local file systems: i nodes 3

3.2. Network file systems: Network File System 6

3.2.1 Partitions and mounting points 6

3.2.2 Client and server functions 9

3.2.3 NFS Elements 9

4. NFS ON RMI 10

5. DEVELOPING THE PRACTICE 11

Bibliography 12

1. Objectives

- ❑ Understand how NFS works in a real-world deployment.
- ❑ Differentiate the meaning of exporting a directory and mounting a directory.
- ❑ Learn how to export and mount directories.

2. Introduction

In this practice the student becomes familiar with a widely known distributed file system and used as the **Network File System (NFS).** NFS allows you to share files and directories on UNIX systems over a LAN or WAN, where users get localization transparency, that is, they can work **with remote** files as if they were **local.** To get started, the student performs a study of the existing NFS configuration in the work lab. The following two sections focus on explaining the concepts of export and assembly, and how to perform them with NFS on a LAN. In conclusion, the problem of inconsistency in the client cache is studied and in addition, messages that are exchanged between client and server in NFS are observed using the Wireshark free distribution *software* network analyzer.

3. File systems

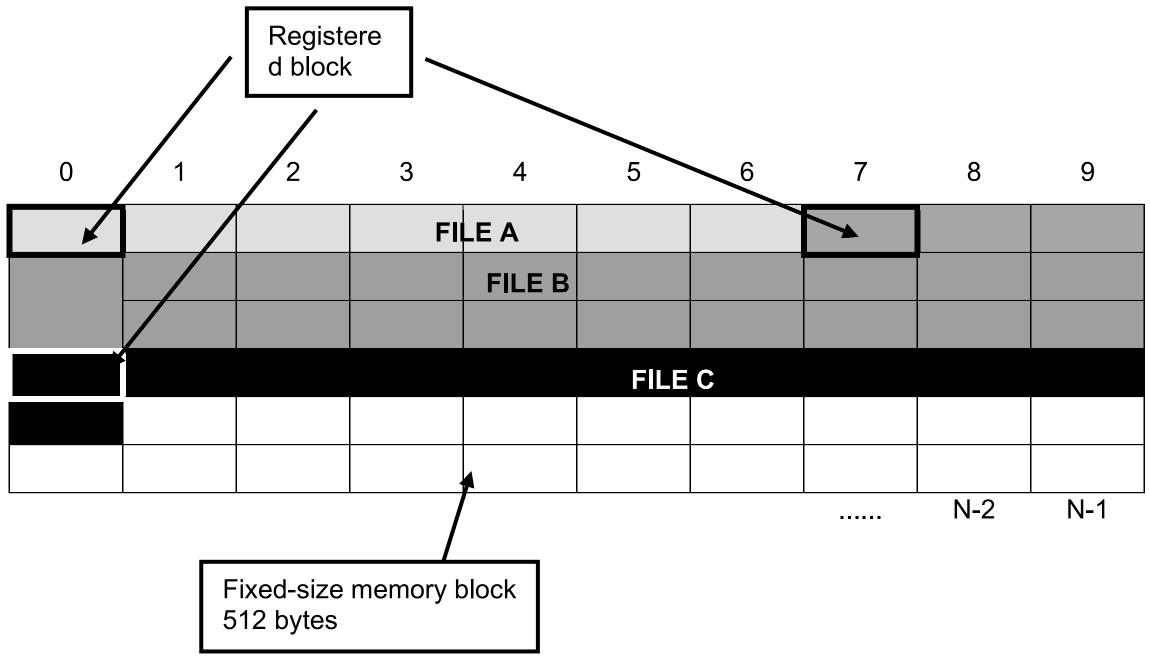
A file can be defined as a sequence of bytes plus attributes (creation date, owner, last modified date, size, and so on). A directory will therefore be a *table* that contains one entry per file, understanding by input a pointer indicating the location of the file. File information (attributes, not content) is stored in directories. The system typically treats directories as files.

On the other hand, a file system will be that part of the operating system that is responsible for the management of files (and directories), including tasks such as creation, modification, reading, writing, change of attributes, etc. The file system typically consists of: directory server and file server.

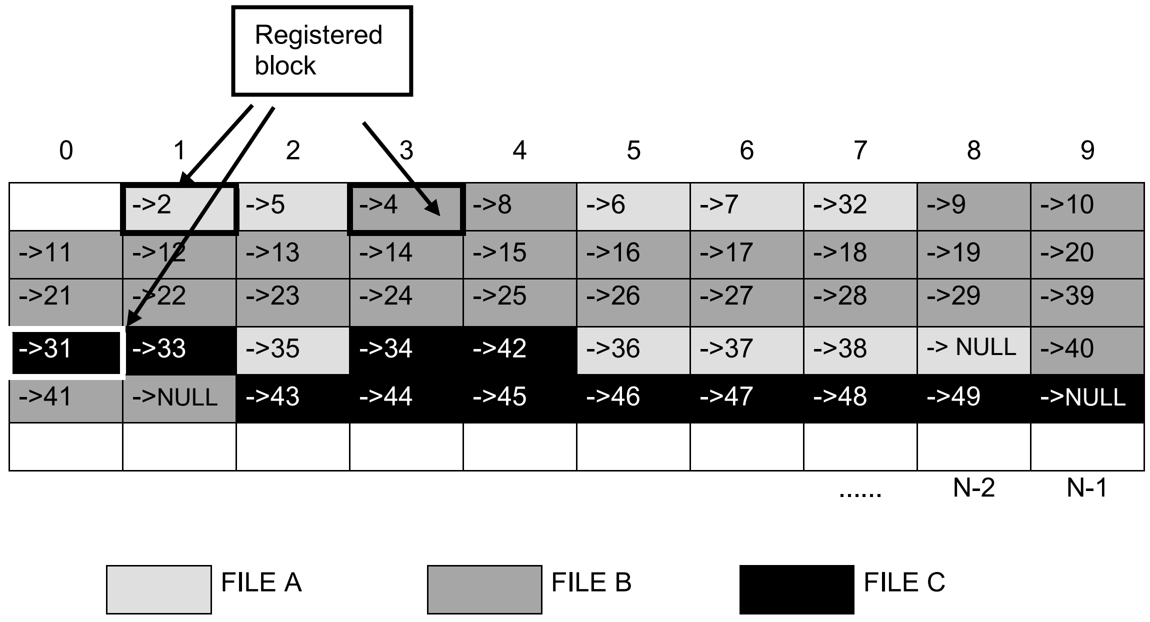
3.1. Deploy local file systems: i nodes

Local file systems can be deployed in several ways:

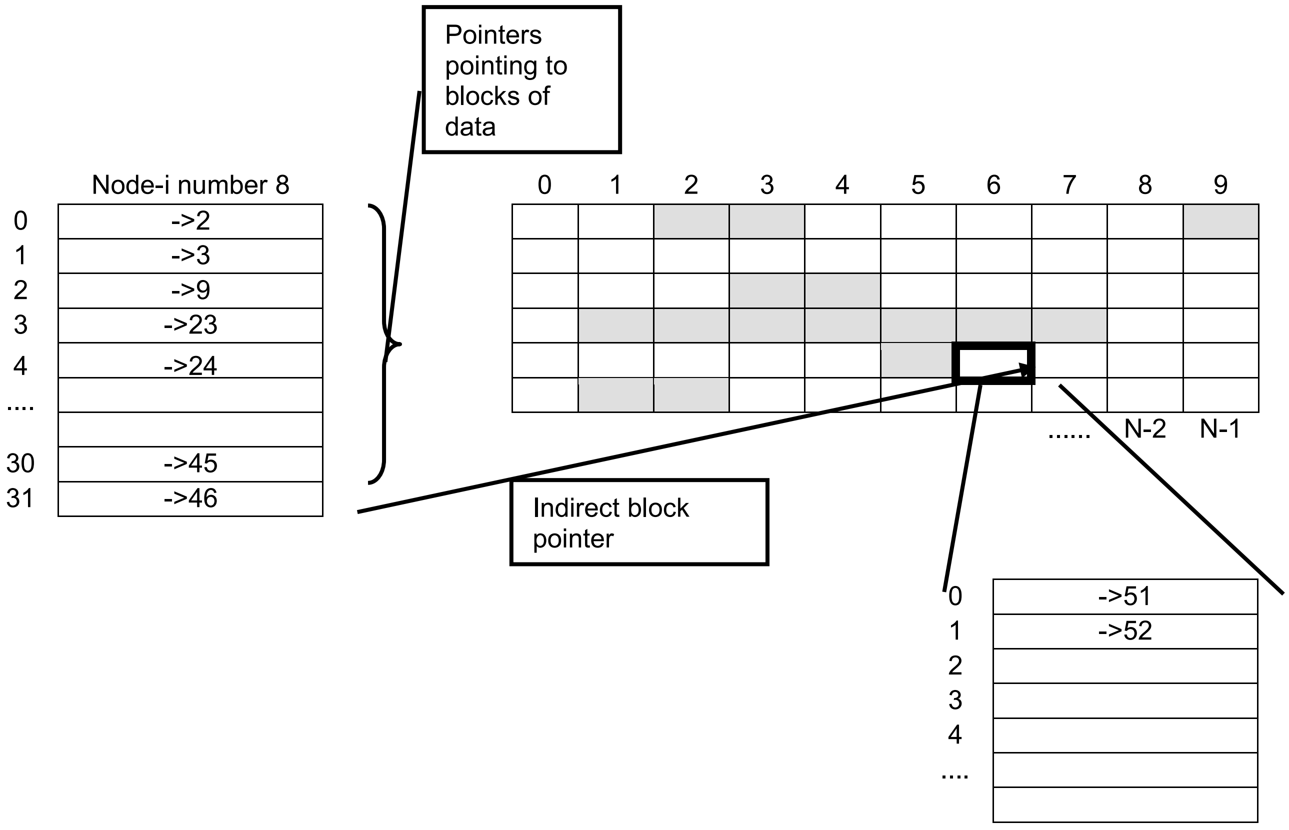
- ❑ *Adjoining assignment*. Each file is stored in an adjoining sequence of blocks on the storage device. The advantages of this implementation are: simplicity, since only the first block and the length of the file are registered, and an easy shortcut, causing high performance if you read entire files. The main problem is what to do if a file grows.



- *Linked list*. Files are stored as a linked list of blocks. In each block the first bytes are used as a pointer and the rest for data. The advantage is that if the file grows after it is created we have no problem. As disadvantages we have that sequential access is not as efficient and random access is slower. In addition, the useful information for each block is no longer two power because the pointer takes up space.

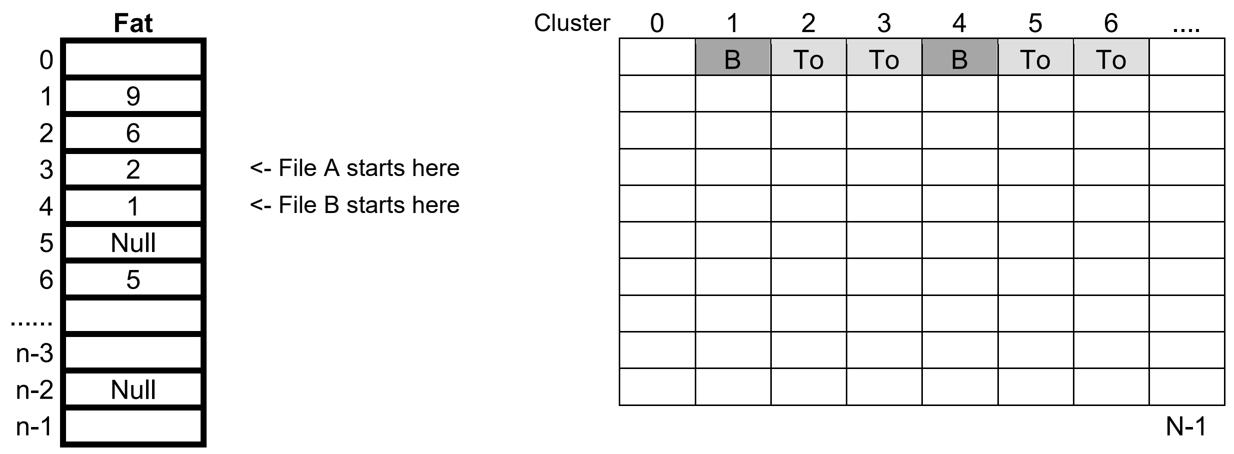


- i-nodes. It is used in the UNIX file system. It consists of keeping together the pointers of each file in a table associated with that file. This table, node-i, is saved in a block where you can also save information about other attributes in the file. The directory in which the file is located will contain a pointer to the block in which the table is stored (pointer to node-i). As advantages we have that only one pointer to node-i is registered, generating a quick random access for small files. The problem is, what to do if the file grows too much and doesn't fit in the table? The solution will be to use a combined schema, data with linked list, being able to have pointers to blocks of data, pointers to simple indirect blocks, to double indirect blocks, etc.; increasing complexity.



Boot	Superblock	BITMAP	LIST i-nodes	LIST BLOCKS DATA
1 block	1 block	N blocks	N blocks	N blocks
Starter code	File system status	Free space management (1 -> busy, 0 -> free)	64x16-bit blocks	Blocks of 512 or 1024 bytes

- FAT (*File Allocation Table*). It is used in its different versions (FAT16, FAT32, VFAT, etc.) on Windows file systems. In this case, all pointers in all files in a table (FAT) are grouped. Instead of saving the data to blocks, clusters (one *cluster* - allocation unit are grouped into a logical one). Each file is occupied by a number of clusters, which do not have to be consecutive. The table contains information from the next cluster in the file, that is, it functions as a linked list, only that the list is implemented in the fixed space allocated to the FAT table. Typically, space is reserved at the beginning of each disk partition for that table, just after the boot sector. FAT itself manages free space. As advantages, any file can grow easily, random access is fast and the information in each block is power of two. As a result, the loss of space dedicated to the FAT table itself.



3.2. Network file systems: Network File System

A distributed file system is one in which clients, servers, and storage devices are scattered across a network, but where access is done transparently to clients. The Network File system, NFS, was created in 1985 by SUN as a distributed file system. In addition, it was designed based on a client/server model, where the SERVER is the machine that EXPORTS a set of files and the CLIENT is the machine that accesses those files. The same machine can act as a client and/or server for different file systems.

The objectives of the NFS design were:

- Transparent access to remote files.
- *Stateless*, the server does not need to save status information for an NFS transaction (open/CLOSE operation does not exist). If the server drops, clients do not have to perform any recovery action, they simply retry until the server or network is reset. NFS version 4 changes this behavior and introduces state-related operations,
- It is not limited to a single operating system. It is designed to support UNIX semantics, MS-DOS, etc.
- The protocol does not depend on special hardware.
- It is independent of the transport protocol (TCP, UDP, ...).
- Access control and protection by UNIX semantics.
- Simple server or client crash recovery mechanism.
- Clients and servers communicate via RPC.

3.2.1 Partitions and mounting points

Locally we can have one or more disks and each of them partitioned (divided) into logical parts. On UNIX almost twosystem components are treated n as a file. Therefore, each partition is assigned special file in the /dev directory that describes it and allows the OS to work with it. These files represent **the physical devices.**.

The **OS** mounts partitions to a particular directory on the file system, which do not match the physical description in/dev. The mount **point is** a particular directory that is used as an entry point to the root file system of the mounted disk or device.

In the example in Fig. 1, we have a PC with two disks, the first (DISK 0) with two partitions. The first partition on disk 0 contains the root directory. The second partition contains the Windows operating system, which in turn is mounted in the root directory */win* of root directory ("/"). Disk 1 has only one partition and is mounted over the */users* directory.

To perform a local mount the mount command **is used,** and the basic command line is:

mount <partition> <mount_point>

For example:

```
SIDusr00@IT5-PC02:> mount /dev/fd0 /mnt
```

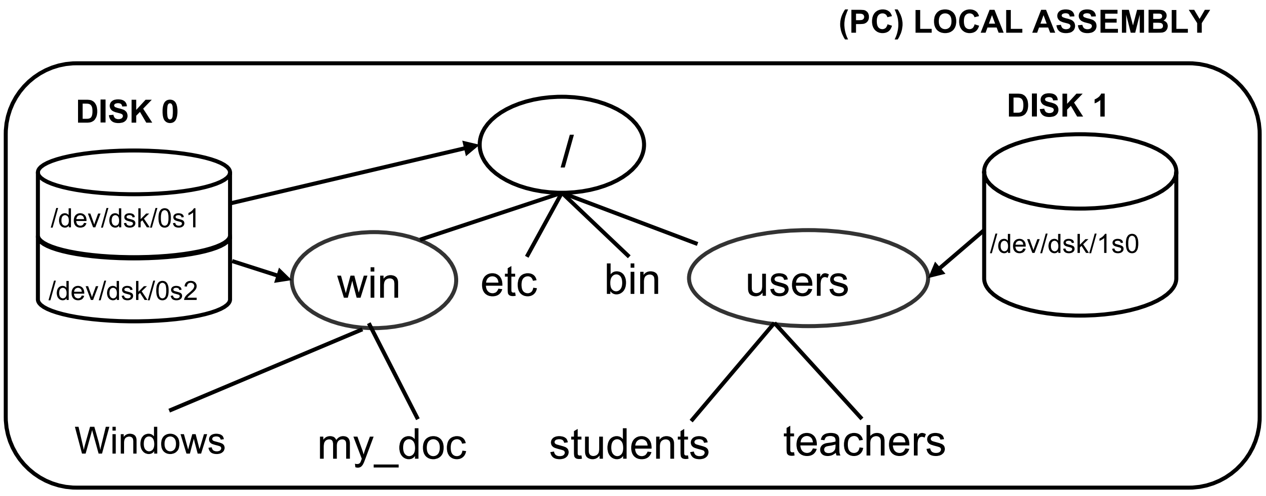
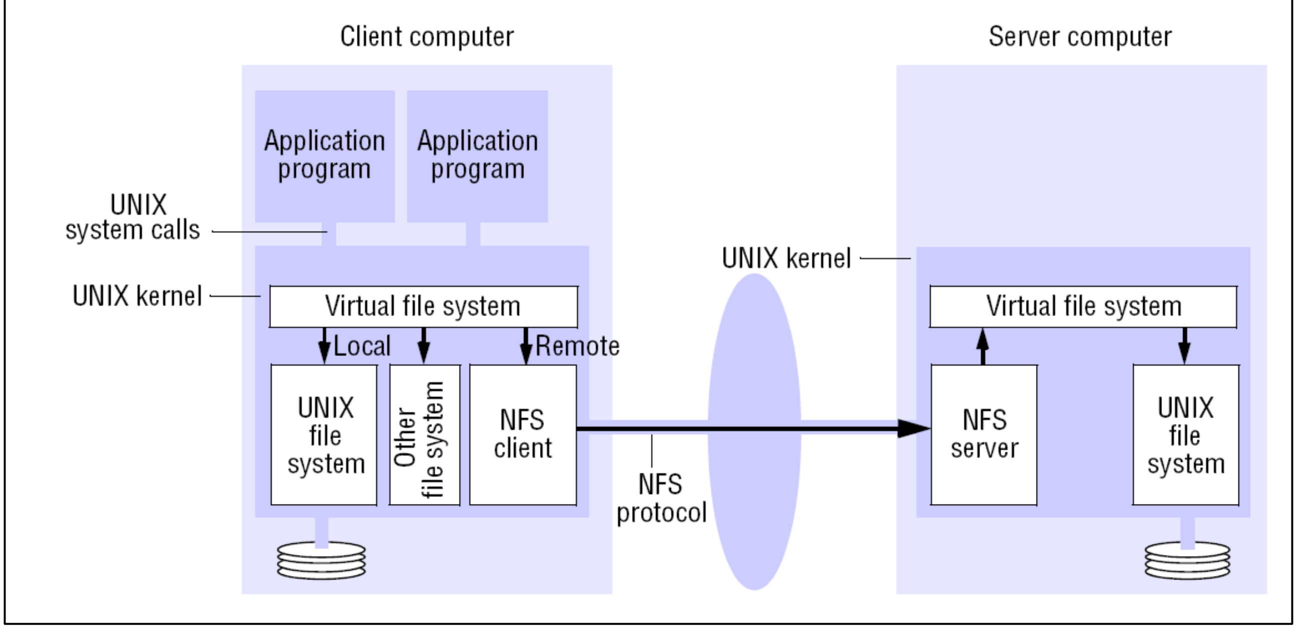


Fig. 1. Example of disk partitions on a machine.

The same idea of local mounting applies to the remote. Let's say we have six machines, IT5-PCxx (where xx goes from 01 to 06) plus the labit501 machine. The capacity of the labit501 machine is much higher than that of IT5-PCxx machines, so it is in labit501 where we have all the working directories of all users on the network. Typically, when an IT5-PCxx machine is booted *into linux,* the first thing it does is mount users' working directories if configured for this purpose. Thus allowing users access to their files and directories, but without having to have a copy on each machine. Physically each user's files and directories are only on the labit501 machine. Once mounted, access requests (read, write, create, etc.) to the remote file system are made using RPC calls that run over the network. The figure shows the architecture of the NFS system.

Figure 12.8 NFS architecture



SOURCE: Distributed Systems: Concepts and Design. George Coulouris. 5th Ed. Pag. 536.

A variant of the **mount** command is used for remote mounting:

mount –t nfs <server>:/<remote_dir> /<local_dir_mount_pount>

For example:

```
SIDusr00@IT5-PC02:> mount -t nfs labit601.upct.es:/users /users
```

With this command line we are saying that on pc IT5-Pc02 we want to mount *the directory /users* of the labit601 machine in the directory */users* of the local machine using the nfs network file system. To unmount a directory, use the **unmount command followed** by the full path of the directory we want to unmount.

REMOTE ASSEMBLY

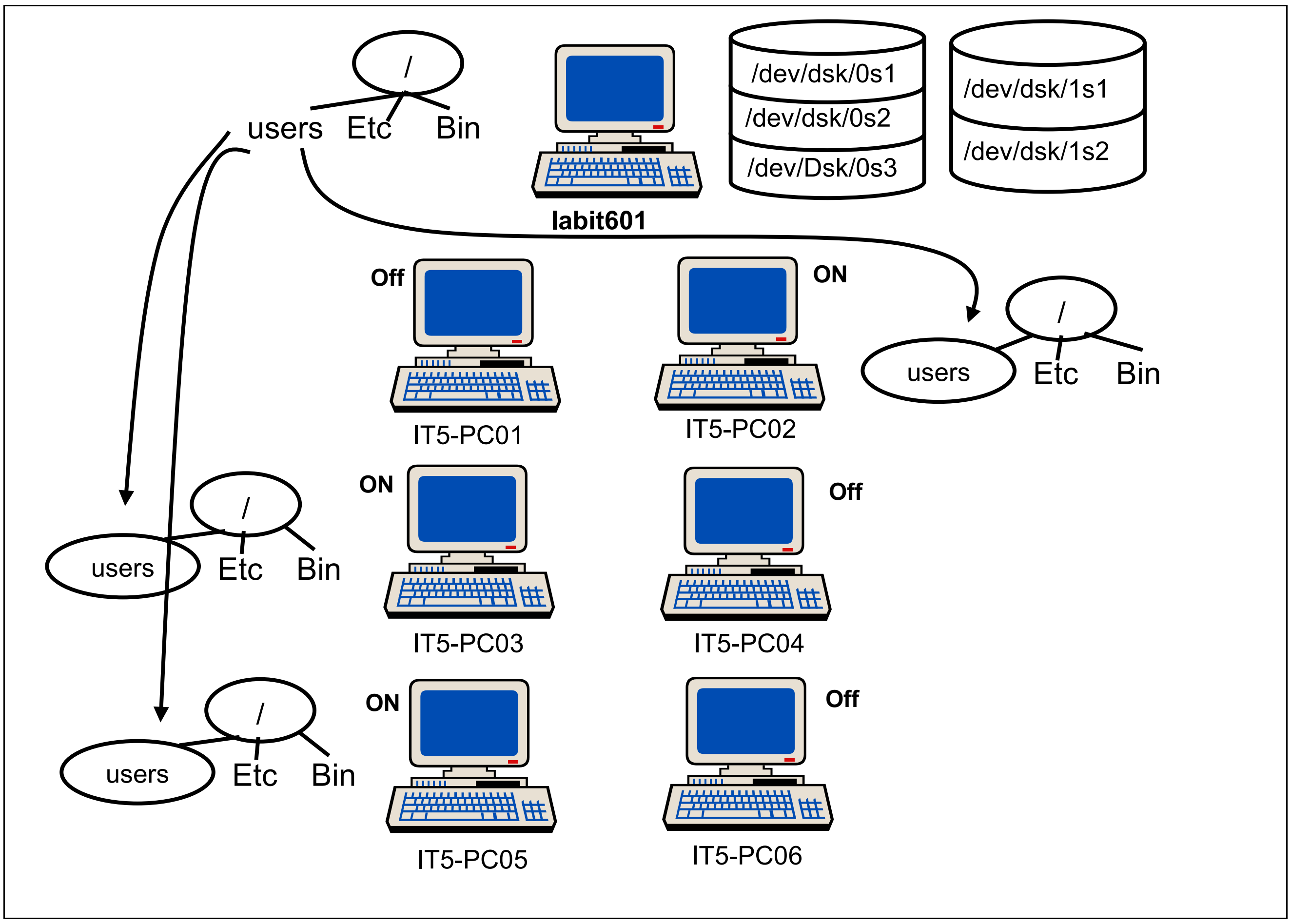


Fig. 2. Example of mounting a directory (/users) from the labit601 server machine to three of the lab PCs. The mount point on each of the client machines is /users.

3.2.2 Client and server functions

The NFS server can export one or more of its directories (including subdirectories). The exported directory can refer to an entire partition or a sub-tree. The server stores in the **/etc/exports file** which clients can access each exported file and the type of access allowed.

The client can mount the file system or a sub-tree of it in its file hierarchy as if it were mounting a local file system. There are two types of mount: *hard*, when any request is retried to the server until it is received and executed, and *soft*, when there is *a time out* after which an error would be reported.

3.2.3 NFS Elements

- *NFS protocol,* which defines the set of requests and arguments that the client can make to the server.
- RPC protocol, which defines the format of client-server interactions.
- XDR, as a machine-independent method for encoding/decoding (serializing/"de-serializing") the data to be sent over the network.
- Mount *protocol* defines the semantics for mounting and dismounting nfs file systems.

- ❑ Demon processes (*daemons*). On the ***nfsd server,***it listens for and responds to requests, and ***mount*d** handles mount requests. On the ***biod*** client, which handles asynchronous input/output.
- ❑ Network Lock Manager (NLM), with ***lockd daemon*** and Network Status Monitor (NSM), ***with statd*** daemon, are used to lock files on a network. With NFS v4 you no longer need to use *statd*.

The server daemon, *nfsd,* will have multiple instances running to get a high degree of concurrency. Each daemon processes a packet from the network and passes it to disk. On small systems there are usually between 4 and 8 simultaneous *nfsd daemons.* Meanwhile, the mountd *server* daemon is responsible for remote mount services, implementing access control. There will only be one instance of it.

The *biod* daemon (or *rpciod*) handles read and write operations and is active on clients. There can be multiple instances of it. The lockd *daemon* (or *nlockmgr*) prevents simultaneous data modifications from multiple clients to the same file, running, as well as a locking system. Finally, the *statd* daemon (or *status*) monitors the nodes in the network and knows the status of a machine.

4. NFS over RMI

Once we have known how NFS works, and seen how RMI works in previous practice, what we will see in this practice is the operation of NFS over RMI. You'll see that the NFS app on RMI will be very simple. As you will remember, in RMI we have a server, where the methods that you want to run are implemented, and a client that sends them to run. Well, in this case, the server will be where the NFS actions are executed, to read and write the files. The client will ask you to access a file from an RMI connection and the server will perform the operations that the client wants to do on that file.

In order for a client to read/edit a file, they must first open the file they want to access. After reading/editing, it is important that the client closes the file that it is accessing.

The server saves to a table (lookup **table**) the files opened for its control,where for each row are saved a unique identifier of the open file, a server identifier, the server time, as well as other attributes (*FileHandler* - file_id + server_id + server_time).

The operations (in a simplified way) that can be done are:

- ❑ **open**(*filename*): The server saves a new row with the FileHandler to its *lookup table.* Returns a number that identifies only the file that has been opened (*file_id*).
- ❑ **read**(*file_id*): The server reads the file with *RandomAccessFile,* either binary, or in characters.
- ❑ **write**(*file_id, new_data*): The server writes to the *new_data* file *with RandomAccessFile,* either binary, or in characters.
- ❑ **close**(*file_id*): The row corresponding to the file with a unique identifier is removed from the lookup *file_id.*

Finally, the general procedure for the operation of NFS over RMI is: once the log is lifted and the server, the client asks to **open** a file, the server saves in its **lookup table** the identifier of the open file, as well as other attributes. The client can now **read/write** to the file, indicating the *id* of the file that the server will have returned to it when opening the file. When the client finishes reading/writing the file, it commands the server to **close** the file, indicating the identifier of the file.

5. Developing the practice

Use Eclipse and Wireshark to develop the following programs:

a) **Develop an NFS server that uses RMI to perform the simplest NFS functions such as reading, writing, opening a file to read/write, and close after reading/writing. These methods will allow you to read and write data in the form of characters.**

- To do this, follow the same architecture as the previous practice, with an interface (*NFS*) where the *open(), close(), read(), and write()* methods *are defined.* Use the *RandomAccessFile class* to get the methods for trading the files.
- Develop a class (*NFSImplementation*) where the methods defined in the interface are implemented. Note that these methods must be defined in the interface in the same way as in the class, so consider which parameters each method has input, as well as the parameters returned by each method. In a simplified way, for the control of open files, we will have two attributes that pretend to be the lookup *table.* In the first attribute named *openFiles we* will save to a Map object Map the name of the open file, as well as its unique identifier. The unique identifier of each file will be determined by the integer hash value of the file name. The second attribute named *filesHandles* will give the identifier of the file and the RandomAccessFile object with which we access the file. It will be with this object that we can read and write from the file, with the methods de of read() and write() that it owns. These functions are detailed in more detail in the RandomAccessFile documentation.
- Finally, develop a client (*NFSClient*), where you open a file, read it, type some text string, and finally close the open file. But is a client leaves a file open, another customer can open or edit the same file.

b) **Develop the previous exercise, but this time trading the data binary.**

c) **Check the operation of RMI via Wireshark.**

- Start the Wireshark program
- To prevent packets that are not of our interest from being included in the capture, include a filter that only captures RMIpackets. At the bottom click on the Filter button. Fill in the Filter Name field with a name for the filter, for example, only_rmi. Then click on the "AddExpression" button and look for the “"rmi.protocol" protocol. Select it along with the “is present" option. Save your changes with Ok.
- Start the capture from the Capture menu and with the Start option. Run one of the two exercisesabove. Stop the capture.
- What messages are exchanged between client and NFS server when we want to see the contents of a file that was previously mounted?
- Re-capture and while the capture is being performed read the contents of the same file. Have the same messages been sent between client and server?

d) **(ADDITIONAL) Implement methods for listing server directories, as well as files within a directory and file properties. In addition, you can create other methods for creating and deleting.**

Bibliography

G.Coulouris, J. Dollimore, T. Kindberg, "Distributed Systems: Concepts and Design", 3rd edition, Ed. Addison-Wesley, 2001.

T. Barr, N. Langfeldt, S. Vidal, T. McNeal, "Linux NFS-HOWTO", <www.tldp.org/HOWTO/NFS-HOWTO/index.html>, August 2002.

"Wireshark", <www.wireshark.com>, 2013.

https://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.htm