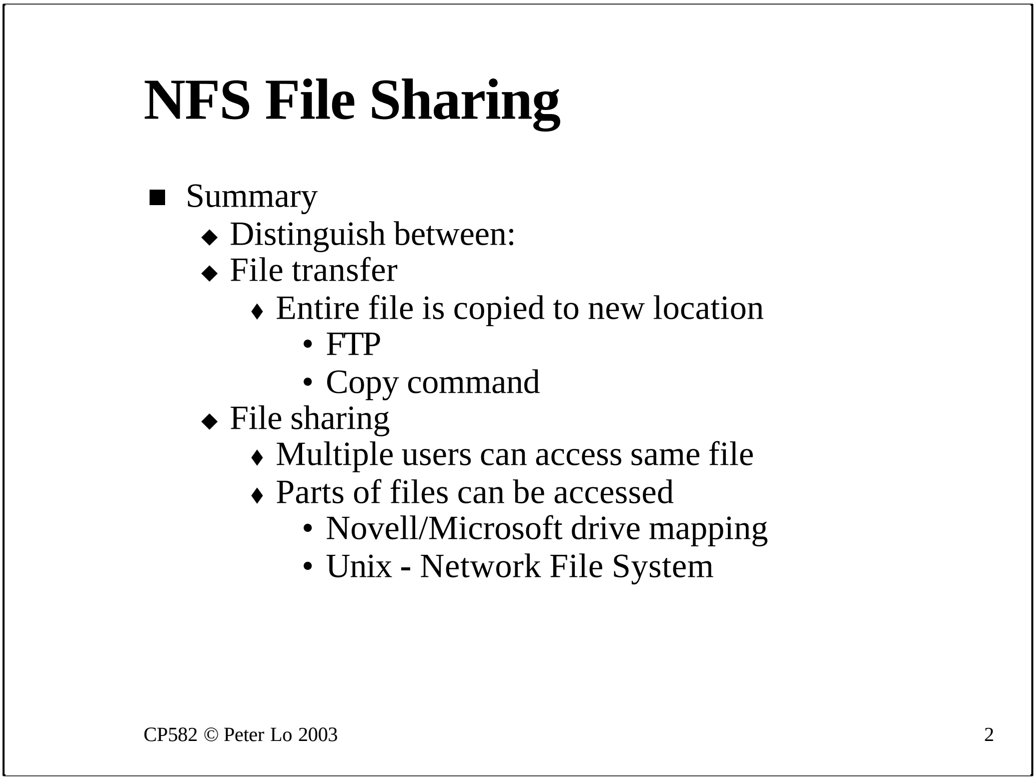
NFS File Sharing

Peter Lo

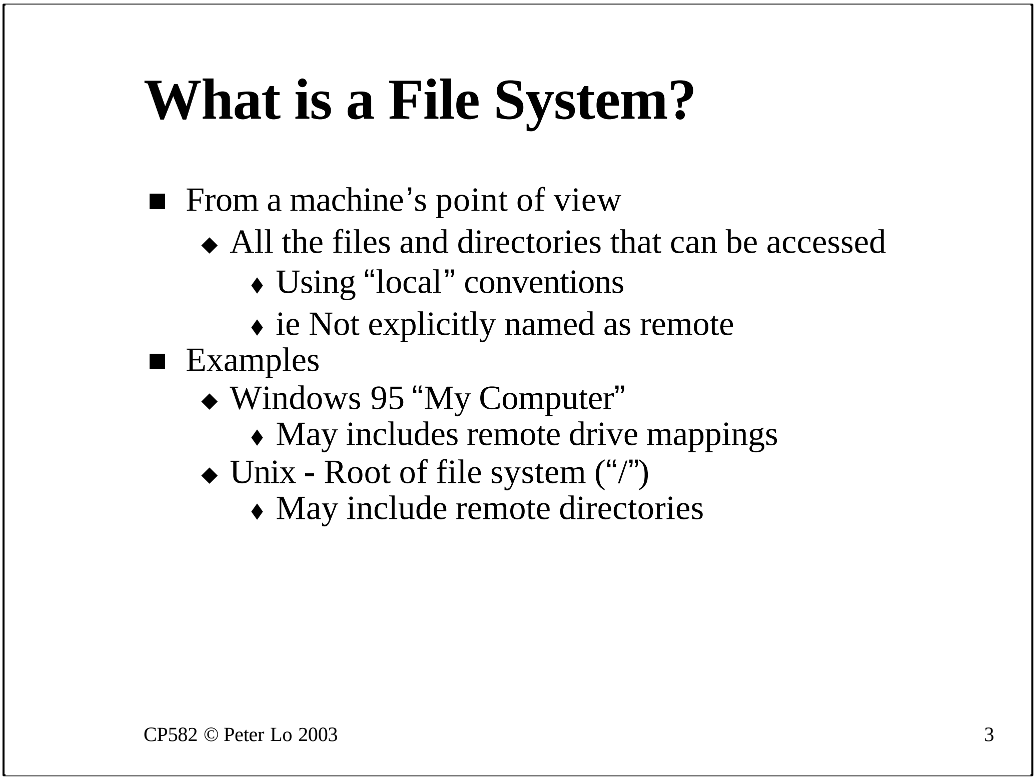


NFS File Sharing

- Summary
  - Distinguish between:
  - File transfer
    - Entire file is copied to new location
      - FTP
      - Copy command
  - File sharing
    - Multiple users can access same file
    - Parts of files can be accessed
      - Novell/Microsoft drive mapping
      - Unix - Network File System

Revision

Just remember the major distinction between file sharing and file transfer.



What is a File System?

- From a machine’s point of view
  - All the files and directories that can be accessed
    - Using “local” conventions
    - ie Not explicitly named as remote
- Examples
  - Windows 95 “My Computer”
    - May includes remote drive mappings
  - Unix - Root of file system (“/”)
    - May include remote directories

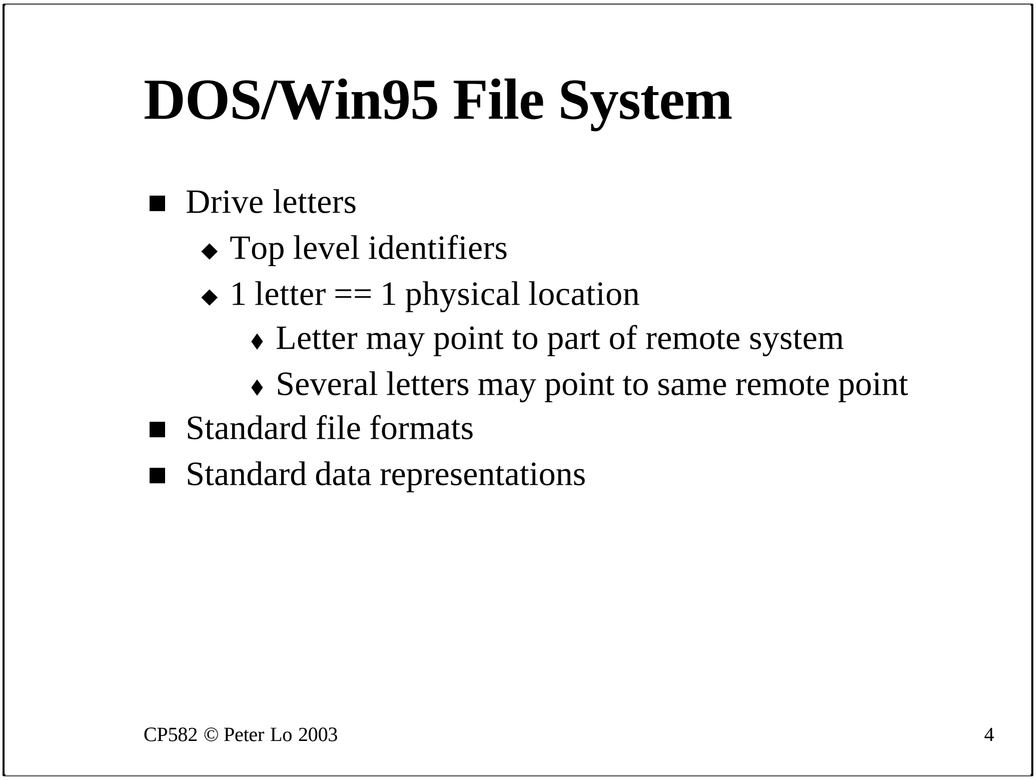
To talk about Unix file sharing we need to have a clear idea of what the term **File System** means.

The **File System** of a computer is the collection of files and directories that can be accessed at that computer.

In the case of Windows 95 the file system is represented by “My Computer” and when we open “My Computer” we see all the drive letters that the computer is currently configured to use. Some of these are local devices (e.g. A: - floppy drive, C: - hard disk , D: - may be the CDROM) but some may be drive letters that have been mapped to remote locations. The drive mapping is the means by which remote **File Systems** become included in our local **File System.**

In the case of Unix exactly the same general concept can be applied. The local **File System**, which we enter via the **root** (“/”) branches to include local devices (e.g. /disk2, /disk7) but may also be configured to include branches that lead to remote **File Systems**. This is where NFS fits in.

To summarise:
W95
My Computer - contains:
        Drive Letters
                Some local
                Some remote
Unix
root - contains
        Directories
                Some local
                Some remote



DOS/Win95 File System

- Drive letters
  - Top level identifiers
  - 1 letter == 1 physical location
    - Letter may point to part of remote system
    - Several letters may point to same remote point
- Standard file formats
- Standard data representations

The DOS/W95 file system uses drive letters as its "top level" branches and we have already seen how each letter can be mapped to a different physical location including the fact that multiple letters can be mapped to the same location.

Once we get to the remote location via the drive letter what we find there is files stored in a standard format that all PCs can interpret.

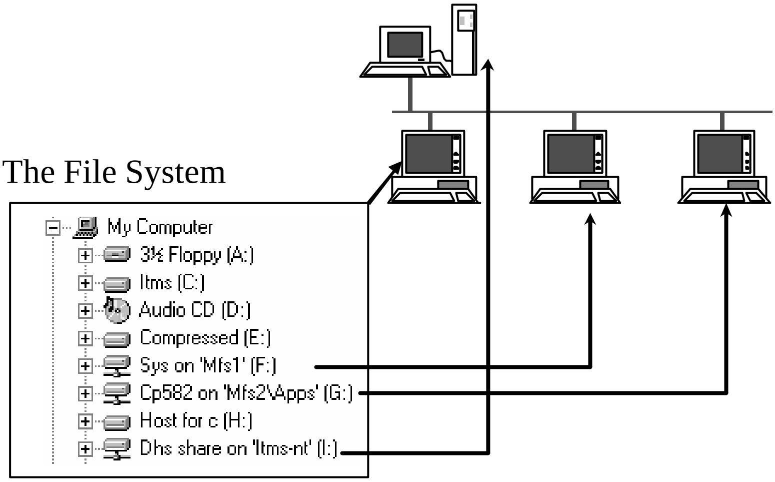
The most obvious example of the **file format** question is the issue of what is used to mark the end of a line of text. There are two possible ASCII characters that can be used to say "move onto the next line" - 0xA which means "line feed" and 0xD which means "carriage return"

If we think for a moment about a physical typewriter the lever that the typist pushes to begin a new line has both these functions - you push is to the right and the paper moves over to the beginning of a new line ("carriage return") and a prolonged push also rotates the drum against which the paper is pressed which moves the paper upwards so that typing begins on a new line ("line feed")

In a DOS text file the beginning of a new line of text is signalled using **both** characters - carriage return, line feed. If we were to look inside the file we would see 0xD, 0xA at the end of each line.

Data Representation considers the issue of how data is likely to be stored in the files. Numbers, for example, should be stored in the same way (at the level of bytes in the file)

DOS File Sharing

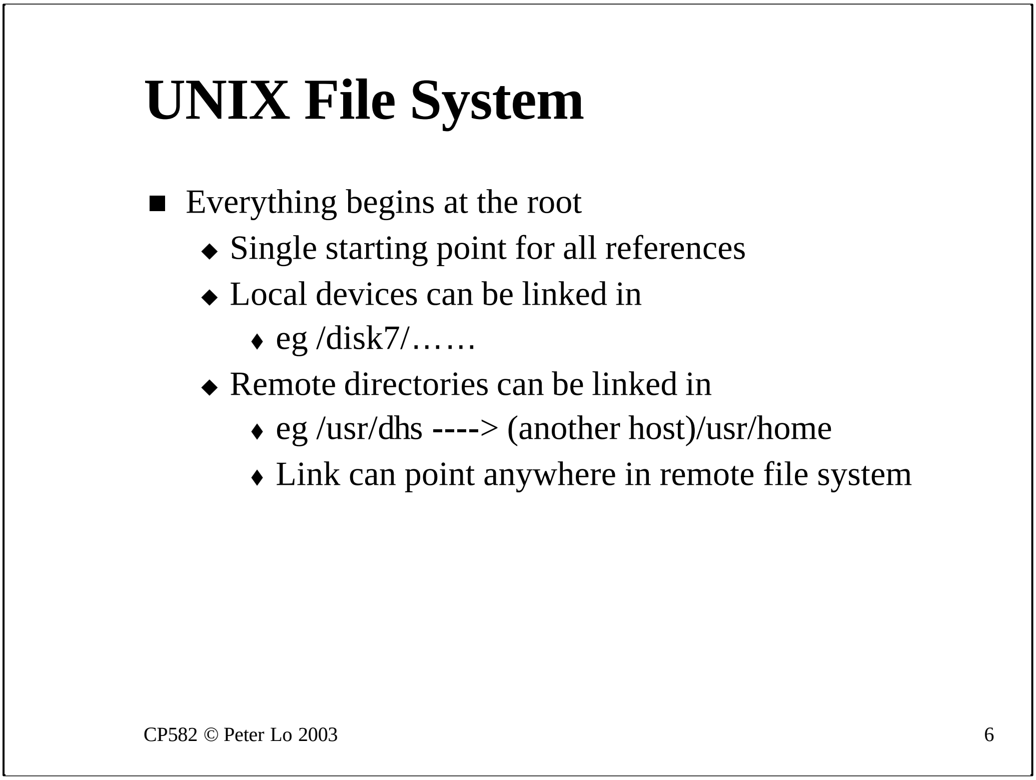


The File System

Here we see the File System of a Windows 95 computer with the drive letters pointing to both local and remote storage locations.

The remote file systems are mostly Novell servers (F:, G:) but one is on an NT server (I:). From the point of view of the Windows 95 computer we are examining though everything, whether provided by Novell or NT is accessed as if were local - using a drive letter.

The level above the drive letters - which has been given the label "File System" in this diagram does not really have a name. Windows 95 calls it "My Computer" but this is just a convenience for the inexperienced user. When we move onto considering Unix we will see that this very top level is given a name - **root** - whereas in DOS each drive letter has a root.



UNIX File System

- Everything begins at the root
  - Single starting point for all references
  - Local devices can be linked in
    - eg /disk7/……
  - Remote directories can be linked in
    - eg /usr/dhs ----> (another host)/usr/home
    - Link can point anywhere in remote file system

Unix has a more unified approach to the concept of File System. The File System begins at the **root** which is represented by the forward slash - **/**

The root is the starting point for all file system references (i.e. names of files or directories that can be accessed)

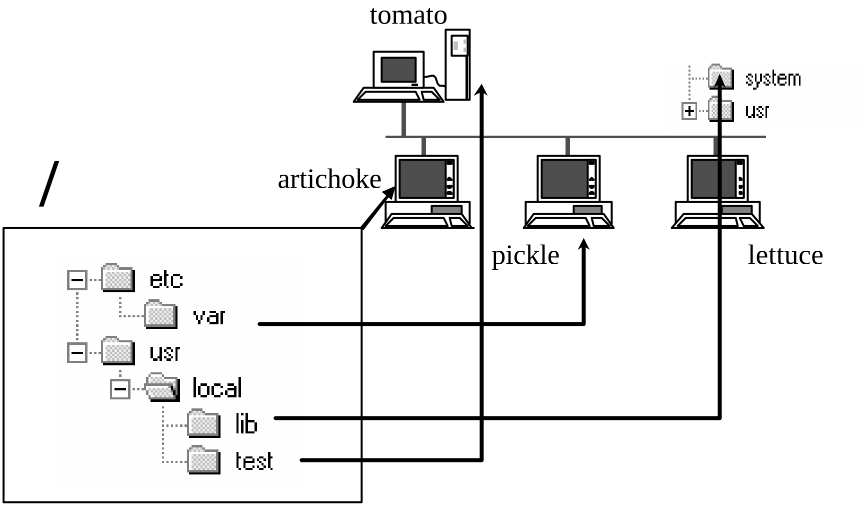
The root itself has to be stored somewhere so there will be a hard disk somewhere that stores the root file system. From this point onwards though the file system can point to storage on other devices.

Some of these other devices may be local but NFS (Network File System) enables remote file systems to be linked to - in order that they appear to be local.

The big distinction between this arrangement and what we saw in DOS is that in Unix you can navigate the entire file system, whether storage is on remote or any local device, simply by using the **cd** command. You can **cd** to one level and be viewing a directory stored locally and then **cd** again and be viewing a directory stored the other side of the world. NFS is the protocol that makes this possible.

Furthermore the linking (unlike drive mapping) can be done at any level in the directory structure - both in terms of the position in the local file system directory layout and the position that is linked to in the remote system.

UNIX File Sharing



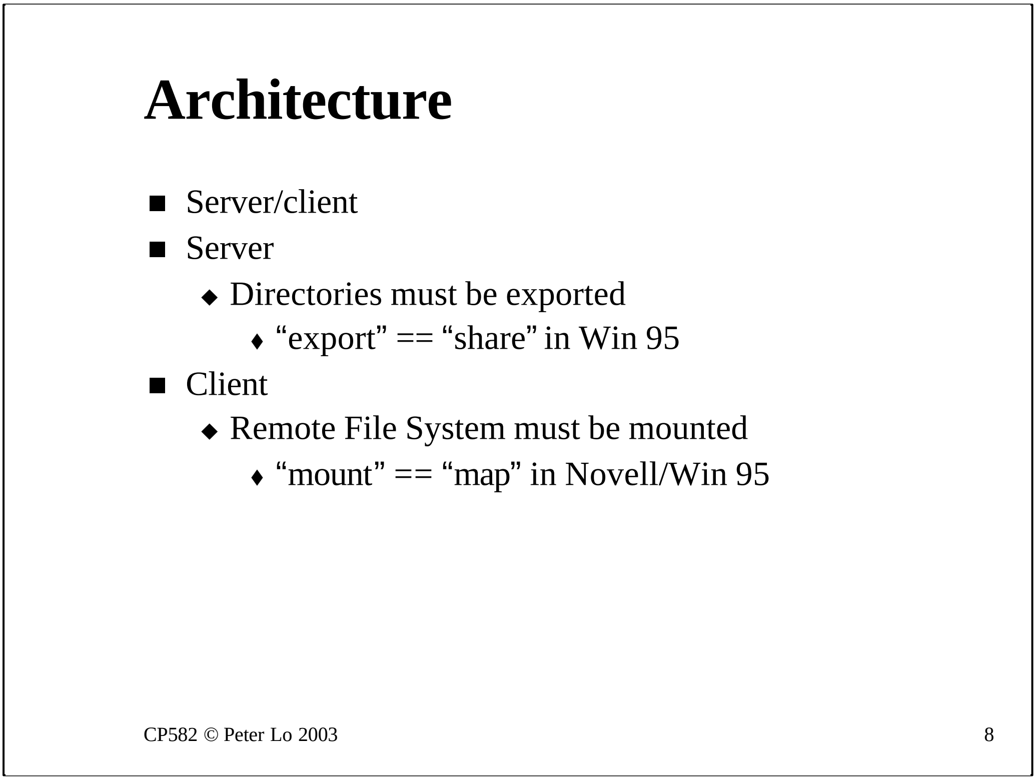
In this diagram we see the reality of NFS linking from the point of view of the File System on a particular Unix host named **artichoke**.

At the top we have the root - /

The directory /etc/var has been linked to a remote machine. Consider starting at the root - **cd etc** will show us local files whereas a subsequent **cd var** will show us files stored on the remote system **pickle**

The directory /usr/local/test has been remotely linked to **tomato**

To make it clear that the any point in the remote system can be linked to it is shown explicitly that /usr/local/lib on **artichoke** is linked to /system on **lettuce**



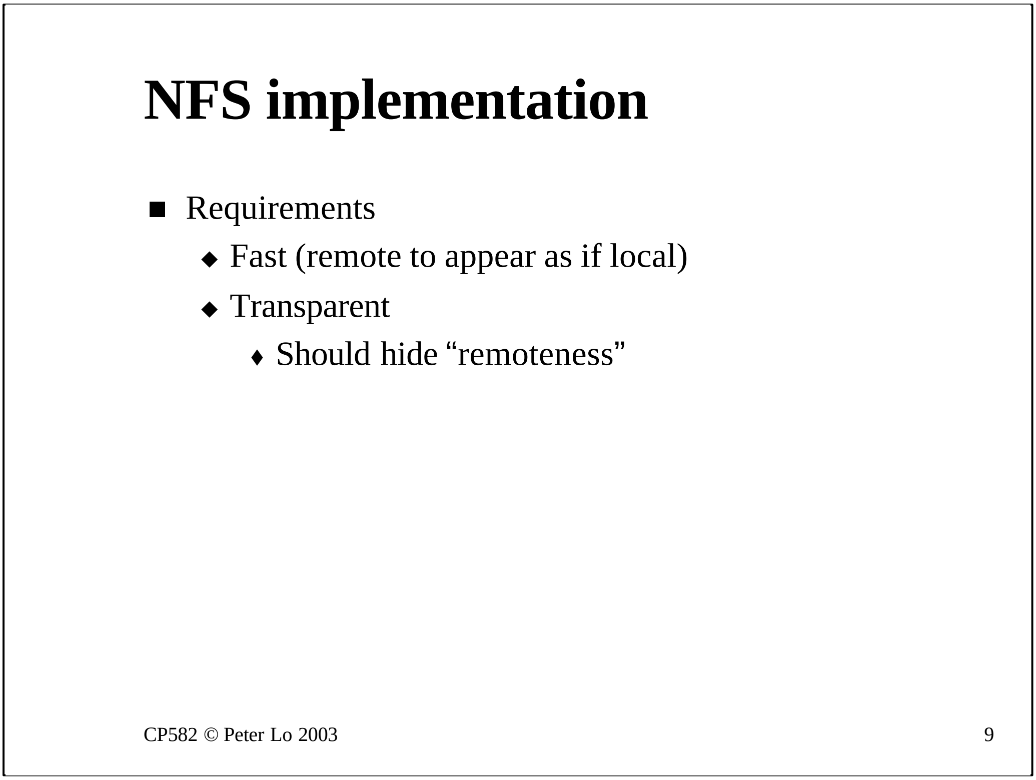
Architecture

- Server/client
- Server
  - Directories must be exported
    - “export” == “share” in Win 95
- Client
  - Remote File System must be mounted
    - “mount” == “map” in Novell/Win 95

Each NFS link has a “server” and a “client” side - of course these roles can switch between one link and the next.

The server side of the link must use a command to state that a particular directory is available for remote access. This is the **export** command which has the same significance as **share** in Windows 95., Notice that both Unix and Windows 95 are file systems that remain local and private until particular regions are shared or exported whereas the file system on a Novell server is fundamentally all shared by definition.

The client side of the link needs the equivalent of the map command - in Unix this means using the **mount** command to state that portion of a remote file system is to be linked into a certain location in our file system.



NFS implementation

- Requirements
  - Fast (remote to appear as if local)
  - Transparent
    - Should hide “remoteness”

The next few slides are going to take us through the internals of NFS - it shows some interesting features that will expose us to some more layers in networking and also get us to start thinking about new ways of accessing remote resources.

This slide summarises the design goals for a system of accessing remote files **as if they were local**

Performance is an issue if the illusion of “localness” is to be preserved.

It is also a fundamental requirement that the remote system **appears to be local**. This illusion is actually taken to quite a low level so that programs written to access local files will continue to work unaltered when the files are shifted to a remote location. The next few slides step aside to look at how this illusion can be created.

Hiding remoteness

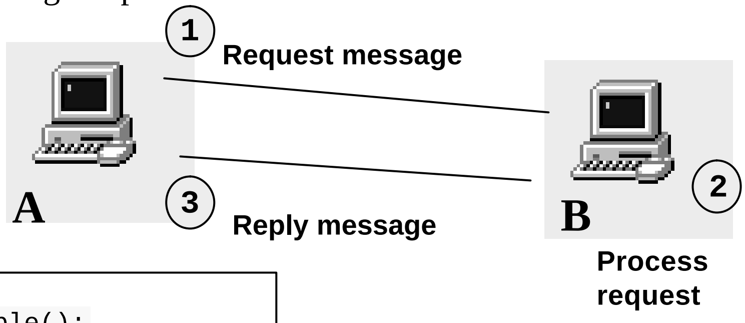
- General propositions
  - Machine A must cause events on Machine B
    - At a program-writing level this means that a routine on “A” must have an effect on “B”
  - Communication channel must exist
    - We have looked at several
      - UDP/TCP
      - IPX/SPX

The question of hiding remoteness begins with the proposition that somehow one machine must be able to cause things to happen on a remote machine. When we access a remote file system there is clearly something happening remotely as a result of something that we do locally.

There needs to be some form of communication between the two machines but we are already experts at that!

A does something on B

- Program point-of-view

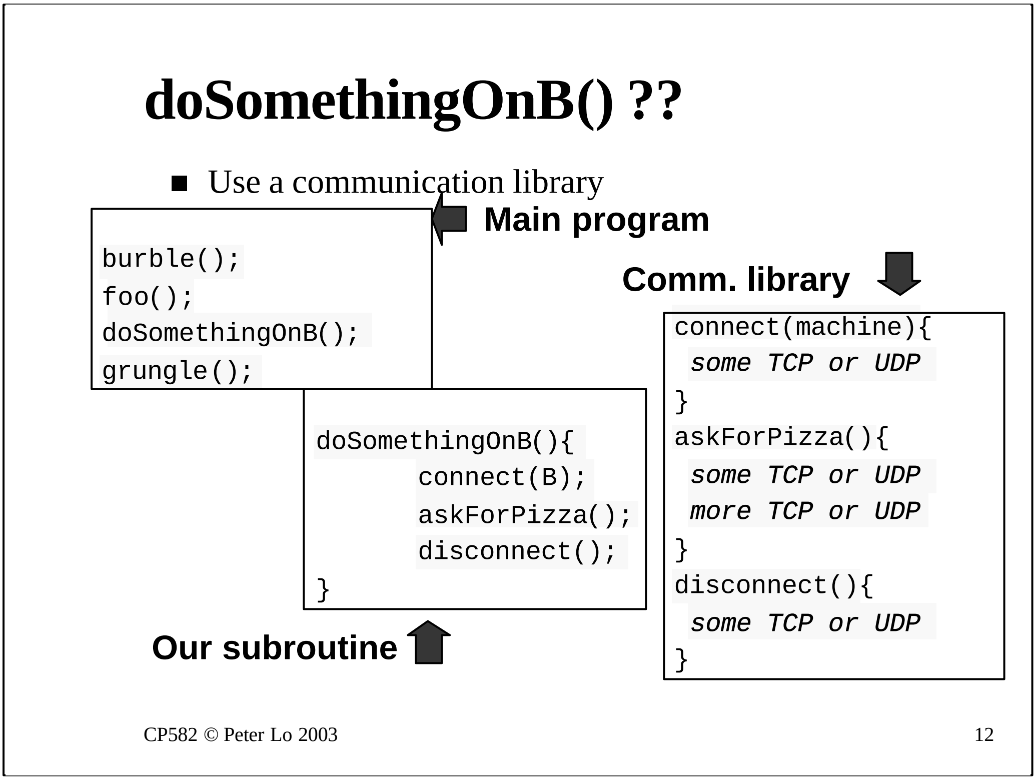


```
burble();
foo();
doSomethingOnB();
grungle();
```

This is a very general look at the proposition that A must be able to cause something to happen on B.

We are looking at things from the point of view of a program that is running at A and we are stepping through the program and seeing when something needs to happen across the wire.

The reason that we are getting down to the level of looking at programs is that when we have finished creating the illusion of "remote as local" we want this illusion to extend right down to the level of programs not caring about remoteness and hence remote files being accessed as if they were local.



doSomethingOnB() ??

- Use a communication library

Main program

```
burble();
foo();
doSomethingOnB();
grungle();
```

Our subroutine

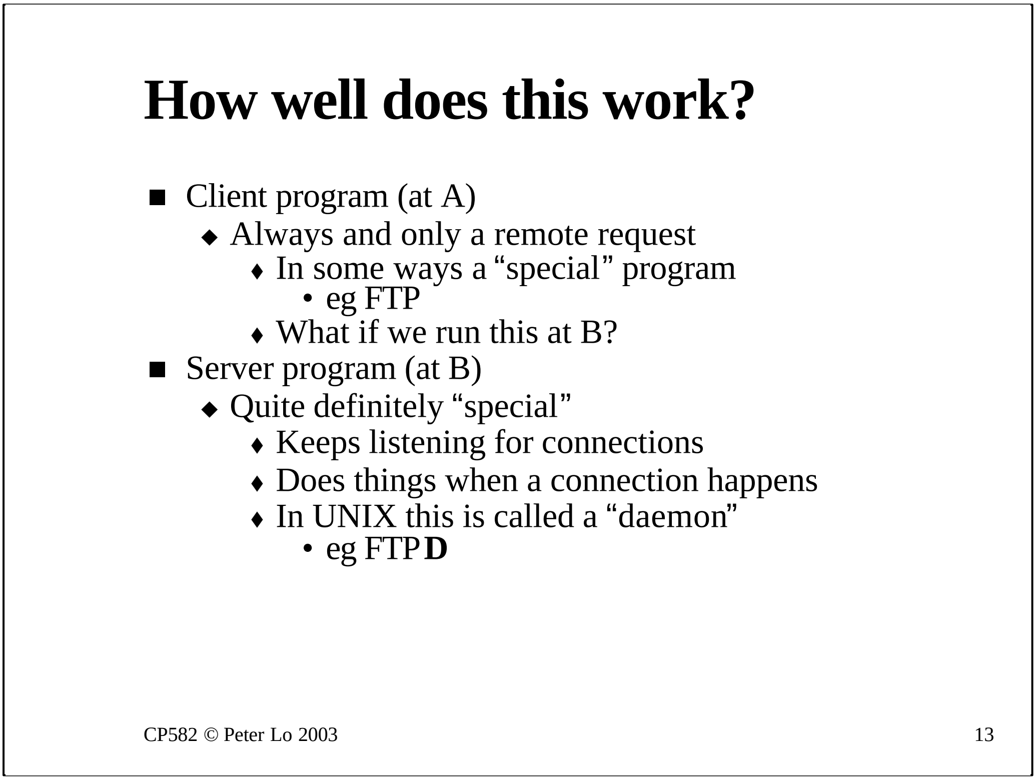
```
doSomethingOnB(){
        connect(B);
        askForPizza();
        disconnect();
}
```

Comm. library

```
connect(machine){
 some TCP or UDP
}
askForPizza(){
 some TCP or UDP
 more TCP or UDP
}
disconnect(){
 some TCP or UDP
}
```

This slide takes things down to the next level and looks at how we might go about implementing something like doSomethingOnB() which needs to communicate remotely in order to work.

The point being made here is that in the course of having an effect on a remote machine there needs to be some low-level communication things (e.g. TCP or UDP) happening and in this example the programmer, who wrote the sub routine doSomethingOnB() needed to know how to use this communication library correctly and has built these calls directly into the sub routine.



How well does this work?

- Client program (at A)
  - Always and only a remote request
    - In some ways a “special” program
      - eg FTP
    - What if we run this at B?
- Server program (at B)
  - Quite definitely “special”
    - Keeps listening for connections
    - Does things when a connection happens
    - In UNIX this is called a “daemon”
      - eg FTP**D**

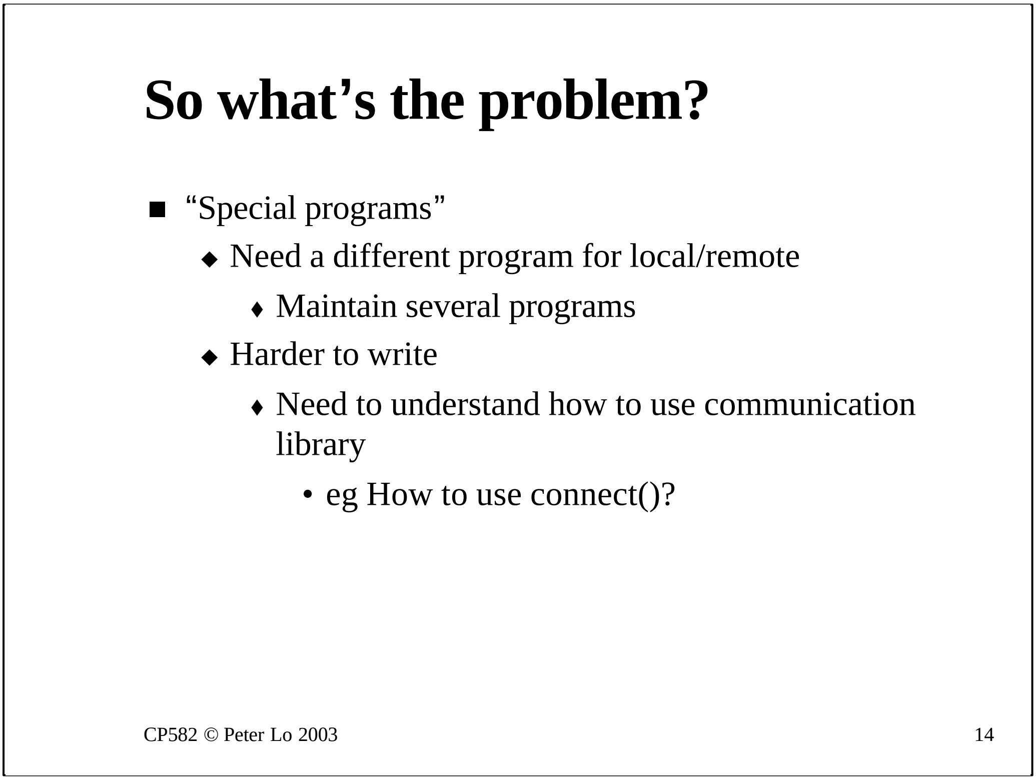
This slide considers the solution so far in a little more depth.

We have pictured a program that has been written in order to do something on a remote computer - to what extent is this program fundamentally “a remote program” - would it still work if the resource (what ever it was) was moved from host B to host A?

The program really has two halves - a client and a server.

In the previous slide we looked closely at the operation of the client side and we saw the special communication library that was used to write the sub-routine. The client program is quite definitely specially written for remote access in the same way that the FTP program that we have run on Windows is.

The server side, which is not pictured in the slide, must be doing something that is quite “remote aware” as well. In general programs that respond to requests from clients must be running in the background on the server host so that they can “wake up” when a request comes in. Such programs, in Unix, are referred to as **daemons** and the server side of FTP is FTPD (the FTP daemon)



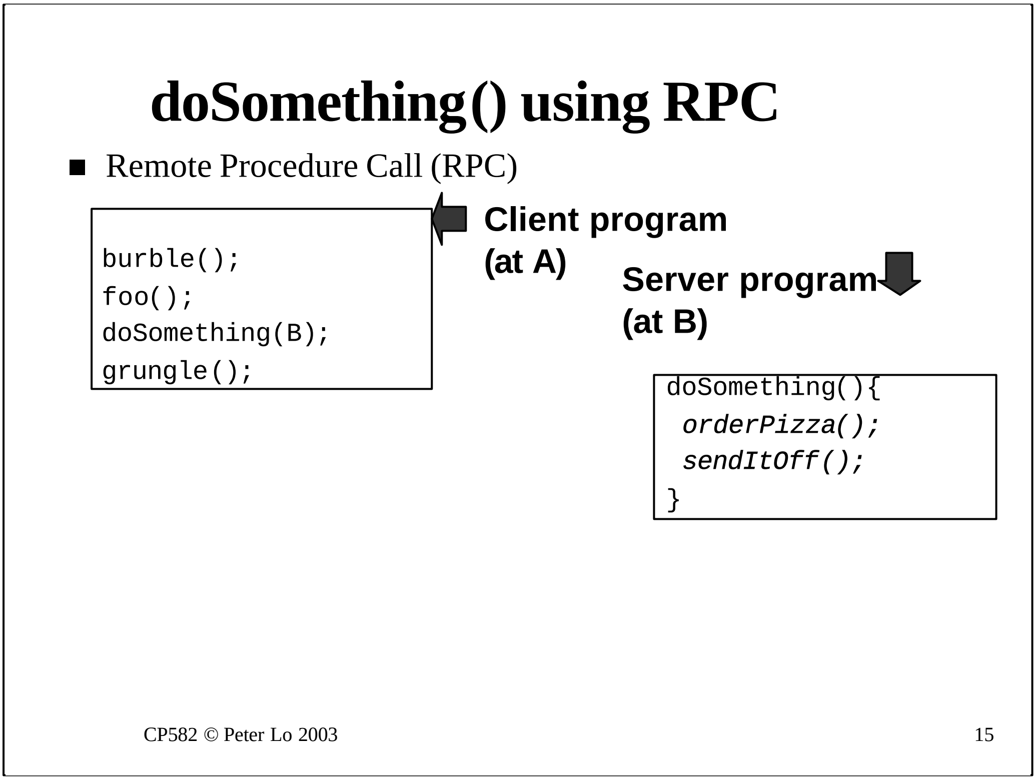
So what’s the problem?

- “Special programs”
  - Need a different program for local/remote
    - Maintain several programs
  - Harder to write
    - Need to understand how to use communication library
      - eg How to use connect()?

In case the previous slide did not make the case strongly enough about “special programs” here are another couple of points to consider:

- If the program has to be specially written for remote access then someone needs to re-write it for local access. This means there are two copies of the program that need to be maintained.

- Writing the program for remote access needs a programmer who understands how to use all the communication bits and pieces - like connect().



doSomething() using RPC

- Remote Procedure Call (RPC)

```
burble();
foo();
doSomething(B);
grungle();
```

Client program (at A)

Server program (at B)

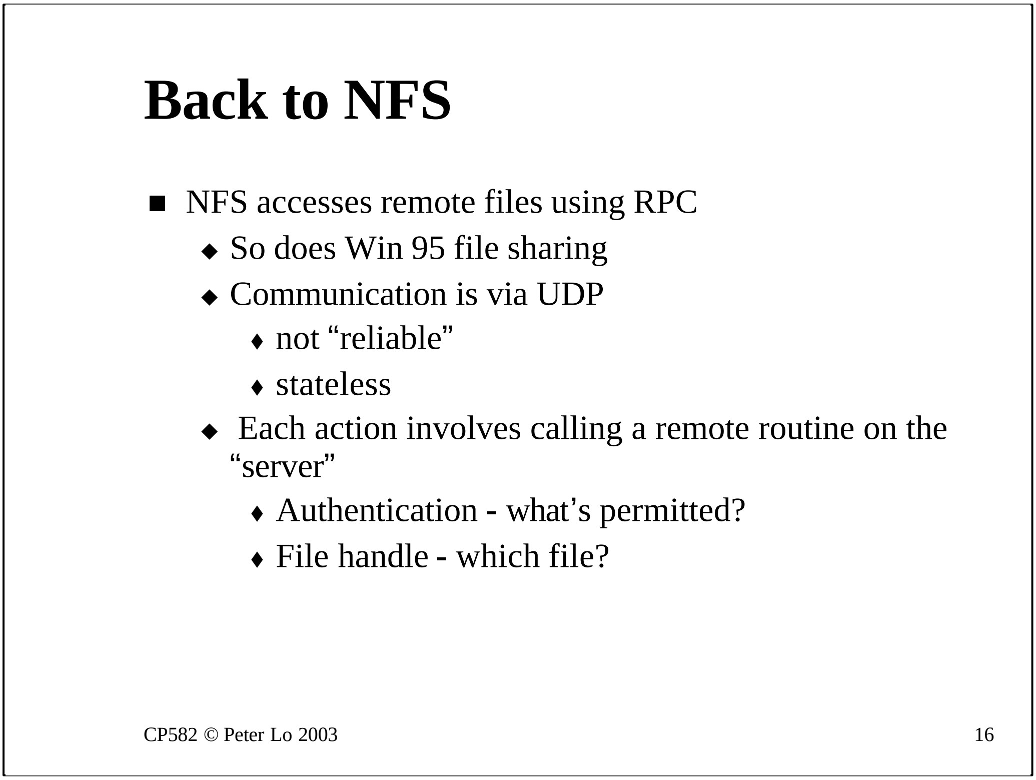
```
doSomething(){
 orderPizza();
 sendItOff();
}
```

What this whole series of arguments is leading towards is the concept of **Remote Procedure Call (RPC)** which is a means for pieces of any program to be transparently executed, if appropriate, on a remote machine.

The program, as far as the person writing the client is concerned, can look as if it is written for local execution - but there are layers within RPC that cause the execution to take place on a remote machine.

In a way this is no big deal compared to what we started out with - there are extra layers of configuration going on at both ends that result in something happening remotely. The key is the extent to which the programmer needs to be aware of all this and the fact that the means for it to happen (RPC) is now something completely generic that all systems know how to do.

For this reason the designers of NFS decided to use RPC to enable the communication of the contents of one file system to a remote machine.



Back to NFS

- NFS accesses remote files using RPC
  - So does Win 95 file sharing
  - Communication is via UDP
    - not “reliable”
    - stateless
  - Each action involves calling a remote routine on the “server”
    - Authentication - what’s permitted?
    - File handle - which file?

NFS uses RPC to get file content from a remote machine and this is a design decision that was shared by Microsoft when they implemented file sharing.

NFS uses UDP - which we will recall is “unreliable”. This has two consequences:

- Performance - UDP, in the absence of errors on the network, will be faster than TCP because there are no sequence numbers and acknowledgments to contend with.

- Stateless - because there is no ongoing connection in UDP there is no way to keep track of “state”. This has some special consequences for NFS.

Each request that the server receives from the client is a unique occurrence - because there is no state. This means that each request needs to have enough information in it to enable the server to sort out the following:

- Does this client have permission to do what they are asking to do?

- What, exactly, are they asking to do? Which particular bit of which file are they wanting to read or write?

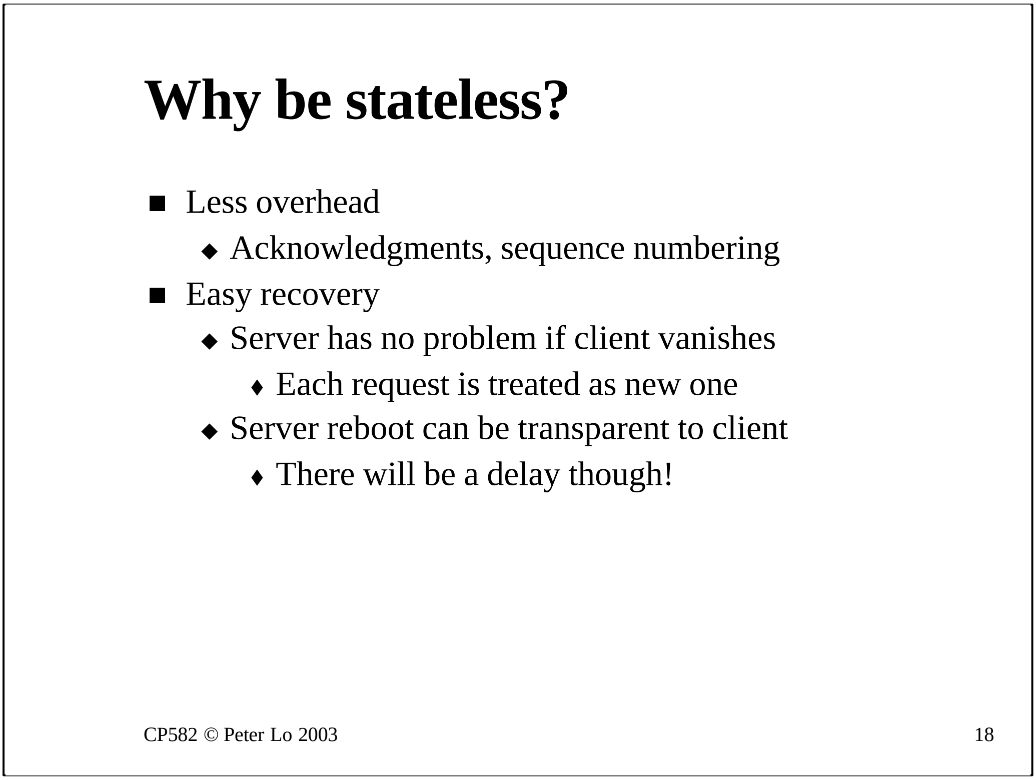
On being Stateless

- A reliable protocol (TCP) has a state
  - eg Connected
  - eg Part way through - sequence numbers
- An unreliable protocol (UDP) has no state
  - Each message is a unique occurrence and is not connected to any other message

Statelessness comes down to the protocols that are used for the communication between the client and the server.

If a reliable protocol like TCP is used then the sequence numbering enables the client and the server to agree about the state of the transaction. The client may say "I'm half way through this transfer" - it is the sequence number that enables the server to know that this is the current state of things.

A protocol such as UDP has no built in ability to share state between the client and the server. Each request that arrives via UDP needs to stand on its own and not depend on any previous request.



Why be stateless?

- Less overhead
  - Acknowledgments, sequence numbering
- Easy recovery
  - Server has no problem if client vanishes
    - Each request is treated as new one
  - Server reboot can be transparent to client
    - There will be a delay though!

At first sight the use of an unreliable protocol might seem to be a real disadvantage but consider the following:

- Performance - UDP has very low overhead. There is no bandwidth being consumed by acknowledgments and sequence numbers and the large headers employed by TCP. Virtually the entire bandwidth being used consists of useful data!

- Recovery - both from server and client failure. From the client's point of view each request stands in its own right so a given request can be satisfied just as well before or after a crash. From the server's point of view a partially completed client request does not imply any overhead in terms of re-establishing contact with the client.

What’s the downside?

- Need very complete information in each request
  - File handle (given out by server) represents location in file system
  - Read/write requests must specify exact offset
    - Cannot say “read next 10 bytes”

There are drawbacks to the “unreliable”scenario in that the information necessary for each request to stand independently makes the requests themselves more complex.

In the case of file access in NFS there are two levels of information that need to be included in every request:

Which file?

This information is given in terms of a **file handle** that uniquely identifies the file. The file handle is issued by the server when the file is first accessed and must be included by the client in all subsequent access to the file.

Where in the file?

The **file offset** is a vital part of any request to read or write parts of a shared file. In conventional file access it is possible to ask for the “next 10 bytes” of a file. Clearly this cannot be part of a stateless system - every request must include the complete file offset.

Authentication

- For entire "session"
  - Takes place when remote system is mounted
- For each operation
  - Need encrypted credential in each request
    - Secure RPC

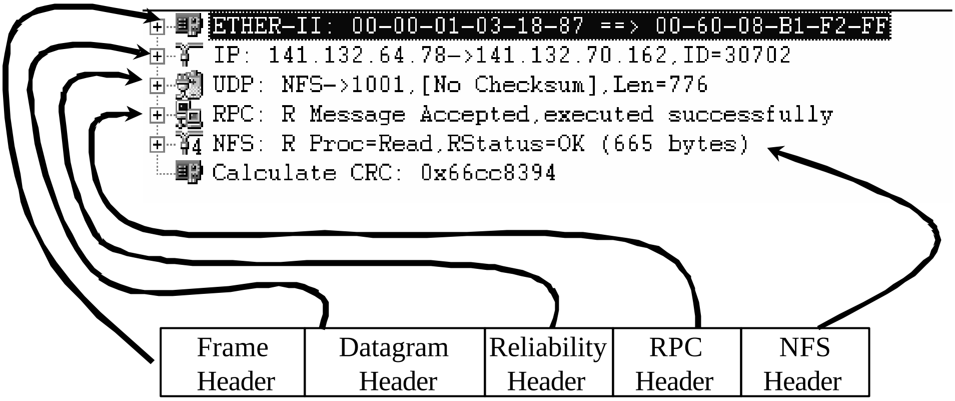
An obvious question to ask is "what about security?"

There are two possible levels of security in NFS - per-session security and per-request security.

At face value per-session security does not seem to make much sense in a protocol that is stateless - there is no concept in NFS file access of a session so how can we make a session secure? The answer is that the security is applied before the file access begins. At the point where a client attempts to **mount** a remote file system there can be a series of security checks imposed. Only once the remote system has been mounted can the (insecure) file access begin.

Per-request security is only available in **Secure NFS**. This flavour of NFS uses encryption of requests to make each request secure in its own right.

The NetXRay view



```
ETHER-II: 00-00-01-03-18-87 ==> 00-60-08-B1-F2-FF
IP: 141.132.64.78->141.132.70.162,ID=30702
UDP: NFS->1001,[No Checksum],Len=776
RPC: R Message Accepted,executed successfully
NFS: R Proc=Read,RStatus=OK (665 bytes)
Calculate CRC: 0x66cc8394
```

Frame Header	Datagram Header	Reliability Header	RPC Header	NFS Header

This is the big picture of an NFS packet.

The usual three layers are now being added to with a layer for the RPC side of things and a layer for NFS.

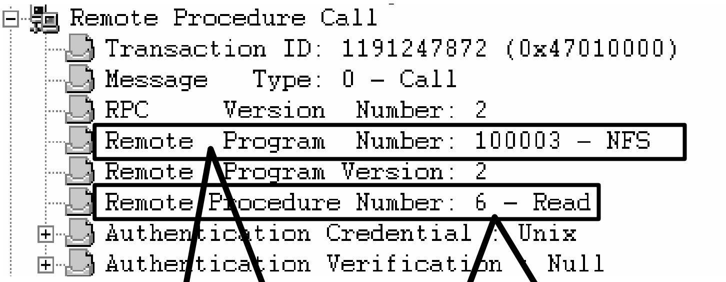
These two layers can really be thought of as part of the Application layer. RPC is a general purpose application that can be used to accomplish any task so long as the correct parameters are passed to the correct server.

In the case of NFS the server is so widely used and so well known that the parameters themselves are being decoded and presented as a separate layer within RPC.

The **five** layers in this packet are the most that we will look at anywhere in NPS.

Reading from a file

We must use RPC to request a read



```
Remote Procedure Call
    Transaction ID: 1191247872 (0x47010000)
    Message   Type: 0 - Call
    RPC     Version  Number: 2
    Remote  Program  Number: 100003 - NFS
    Remote  Program Version: 2
    Remote Procedure Number: 6 - Read
  + Authentication Credential : Unix
  + Authentication Verification : Null
```

Which program do we want to talk to?
NFS is "Program Number 100003" (in the whole world!)

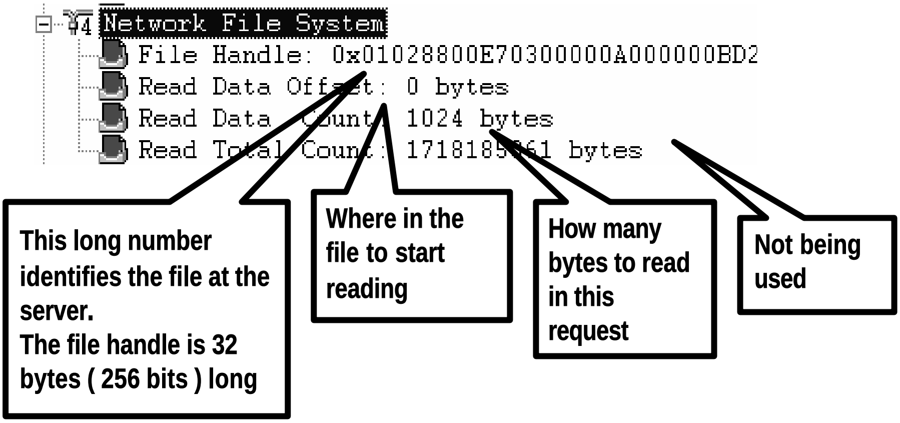
What do we want to ask NFS to do?
***Read* is "Remote Procedure Number 6" (in NFS)**

Here we see that at the RPC layer the request is identified as being an NFS request by means of the "Program Number" field. NetXRay recognises the special value (100003) that has been assigned globally to NFS and hence will proceed to decode the rest of the request as is seen in the next slide.

The NFS server has a number of different sub routines in it that can be remotely executed and the RPC header also needs to indicate which one of these routines is to be executed. In this case it is a Read request.

But <u>what</u> do we want to read?

This is where the NFS header comes in



```
Network File System
    File Handle: 0x01028800E70300000A000000BD2
    Read Data Offset: 0 bytes
    Read Data  Count: 1024 bytes
    Read Total Count: 1718185861 bytes
```

This long number identifies the file at the server.
The file handle is 32 bytes (256 bits) long

Where in the file to start reading

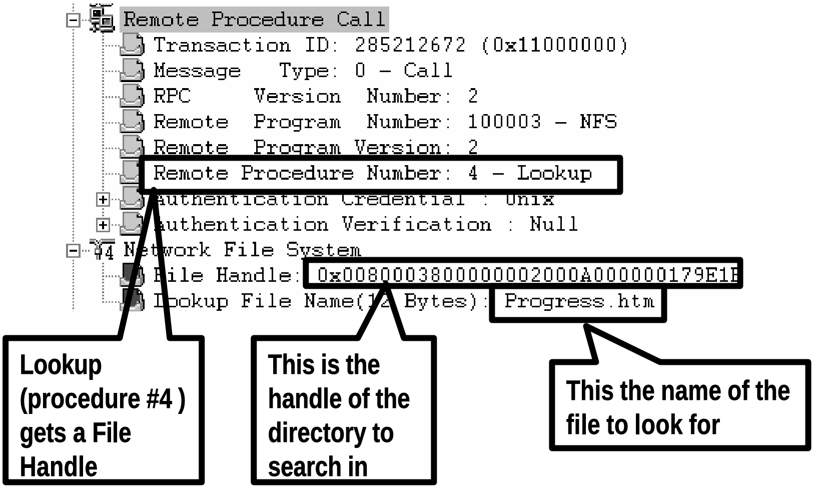
How many bytes to read in this request

Not being used

The “NFS Header” contains information in a format that is specific to this particular NFS request.

In the case of a Read **each request** must uniquely identify the file that is to be read as well as which bytes are to be read.

Getting a Handle..



```
Remote Procedure Call
    Transaction ID: 285212672 (0x11000000)
    Message   Type: 0 - Call
    RPC     Version  Number: 2
    Remote  Program  Number: 100003 - NFS
    Remote  Program Version: 2
    Remote Procedure Number: 4 - Lookup
    Authentication Credential : Unix
    Authentication Verification : Null
Network File System
    File Handle: 0x0080003800000002000A000000179E1B
    Lookup File Name(12 Bytes): Progress.htm
```

Lookup (procedure #4) gets a File Handle

This is the handle of the directory to search in

This the name of the file to look for

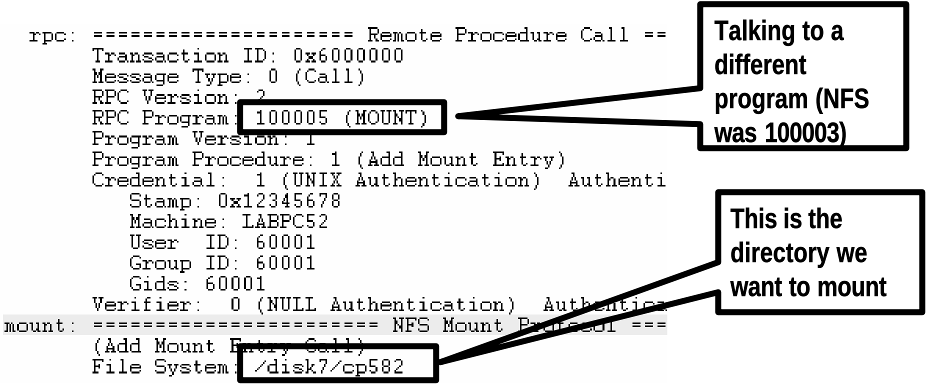
This slide considers a different NFS request - the Lookup request.

The Lookup request is the one that is used to retrieve the file handle that needs to be used for subsequent Read requests.

Lookup is performed relative to **yet another handle** - the handle for the directory that has been shared.

The Directory Handle...

- The handle for the directory is returned by the Mount request
- NetXRay does not know how to decode Mount requests...



```
rpc: ===================== Remote Procedure Call ==
      Transaction ID: 0x6000000
      Message Type: 0 (Call)
      RPC Version: 2
      RPC Program: 100005 (MOUNT)
      Program Version: 1
      Program Procedure: 1 (Add Mount Entry)
      Credential:  1 (UNIX Authentication)  Authenti
         Stamp: 0x12345678
         Machine: LABPC52
         User  ID: 60001
         Group ID: 60001
         Gids: 60001
      Verifier:  0 (NULL Authentication)  Authentica
mount: ====================== NFS Mount Protocol ===
      (Add Mount Entry Call)
      File System: /disk7/cp582
```

Talking to a different program (NFS was 100003)

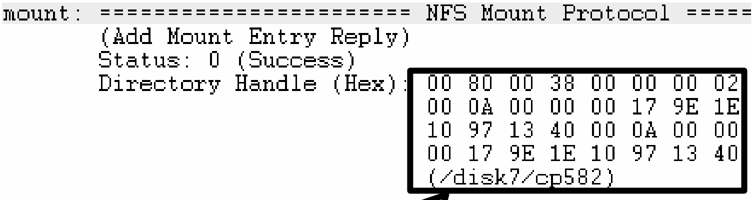
This is the directory we want to mount

(Thanks to Novell's Lanalyzer program)

If we want to trace back to find where the handle for the directory came from we suddenly land up outside of the abilities of NetXRay.

The Directory Handle is returned by a **different remote program** - the NFS Mount program. Mount (as you can see in this capture from Analysers) is remote program 100005 and NetXRay has not been told how to decode 100005 requests.

…being returned



```
mount: ====================== NFS Mount Protocol =====
       (Add Mount Entry Reply)
       Status: 0 (Success)
       Directory Handle (Hex): 00 80 00 38 00 00 00 02
                               00 0A 00 00 00 17 9E 1E
                               10 97 13 40 00 0A 00 00
                               00 17 9E 1E 10 97 13 40
                               (/disk7/cp582)
```

This is the 32 byte handle for the named directory

The end result of the Mount request is that the directory is mounted (assuming that we have permission etc) and that the handle for the directory is returned to the client for use in subsequent requests.