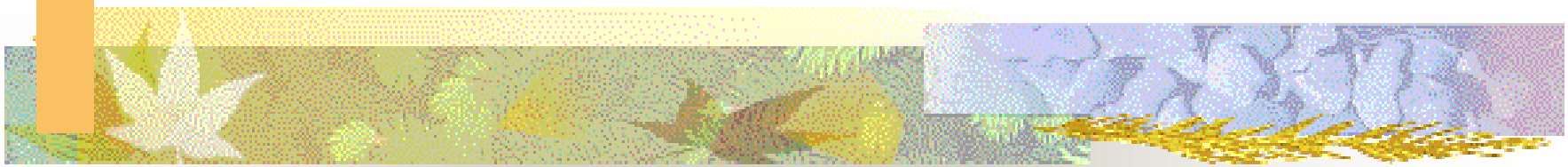
# An Introduction to NFS



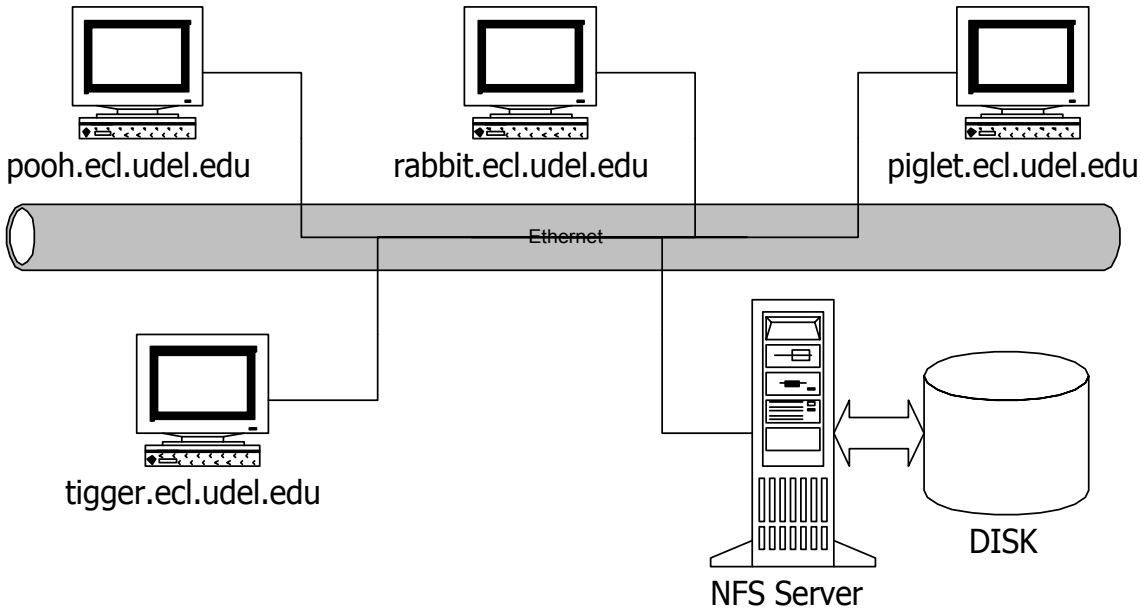
Derived from an article by Frederic Ranal and a presentation by Chavalit Srisathapornphat

# Agenda

- Introduction to NFS
- Introduction to File Systems
- The NFS Protocol
- RPC
- NFS Server Configuration
- Using the NFS Client
- Security Considerations
- Major differences between Versions 2 & 3

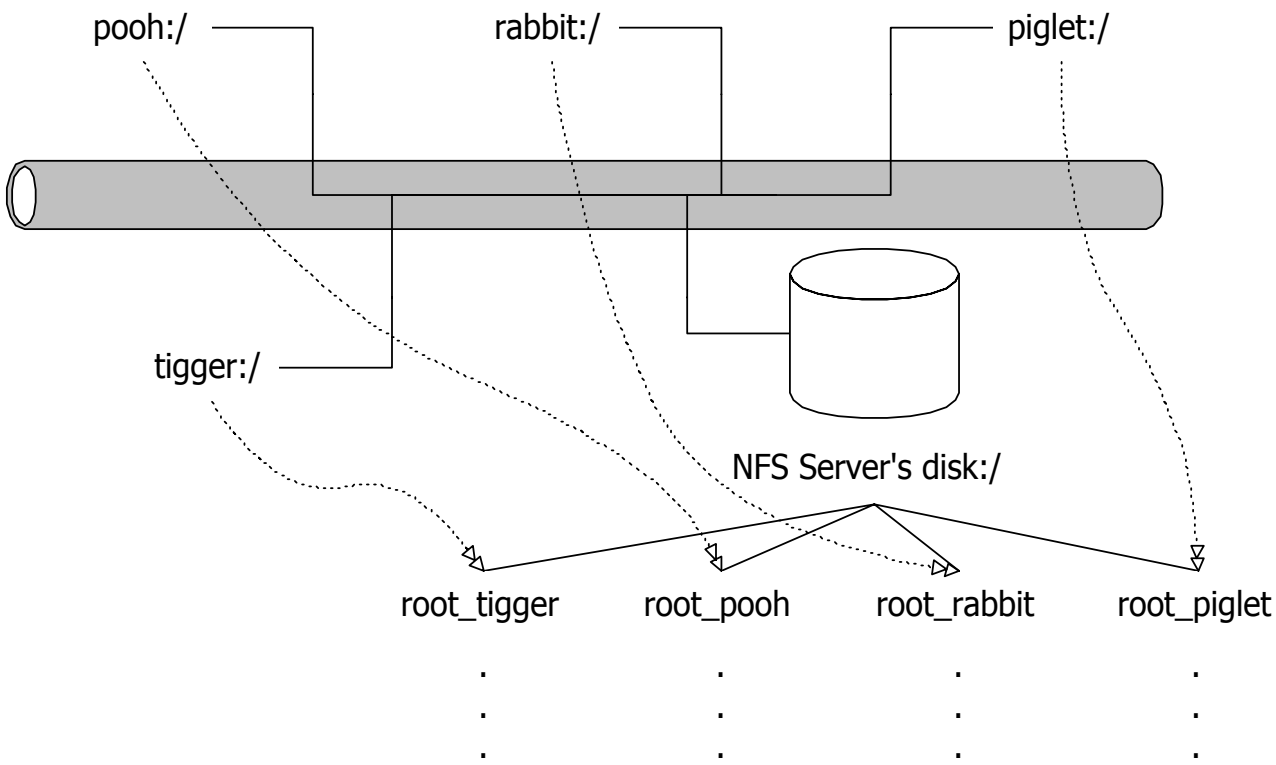
# Introduction to NFS

- Sharing of data between several computers
- A protocol designed for local networks
- Needs to be administered with care



# Introduction to NFS

- Aim: to provide the following logical view

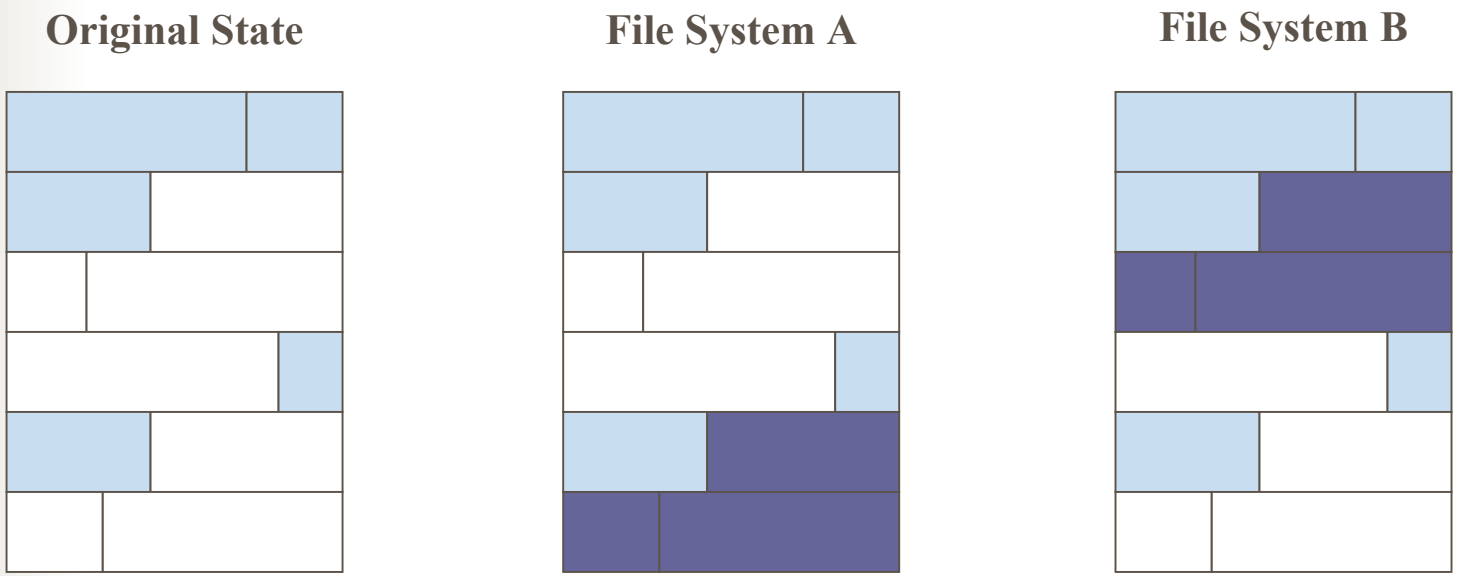


# Introduction to File Systems

- A file system is a way of storing data on a medium: the way it is organized and managed
- Examples: NTFS, HPFS, DOS, FAT, ext2, JFS, ISO9660
- Every media for data can be considered as an array of small units holding information (i.e. blocks)

# Introduction to File Systems

- Every file system manages these blocks differently
- For example, insert a file that will use two blocks:



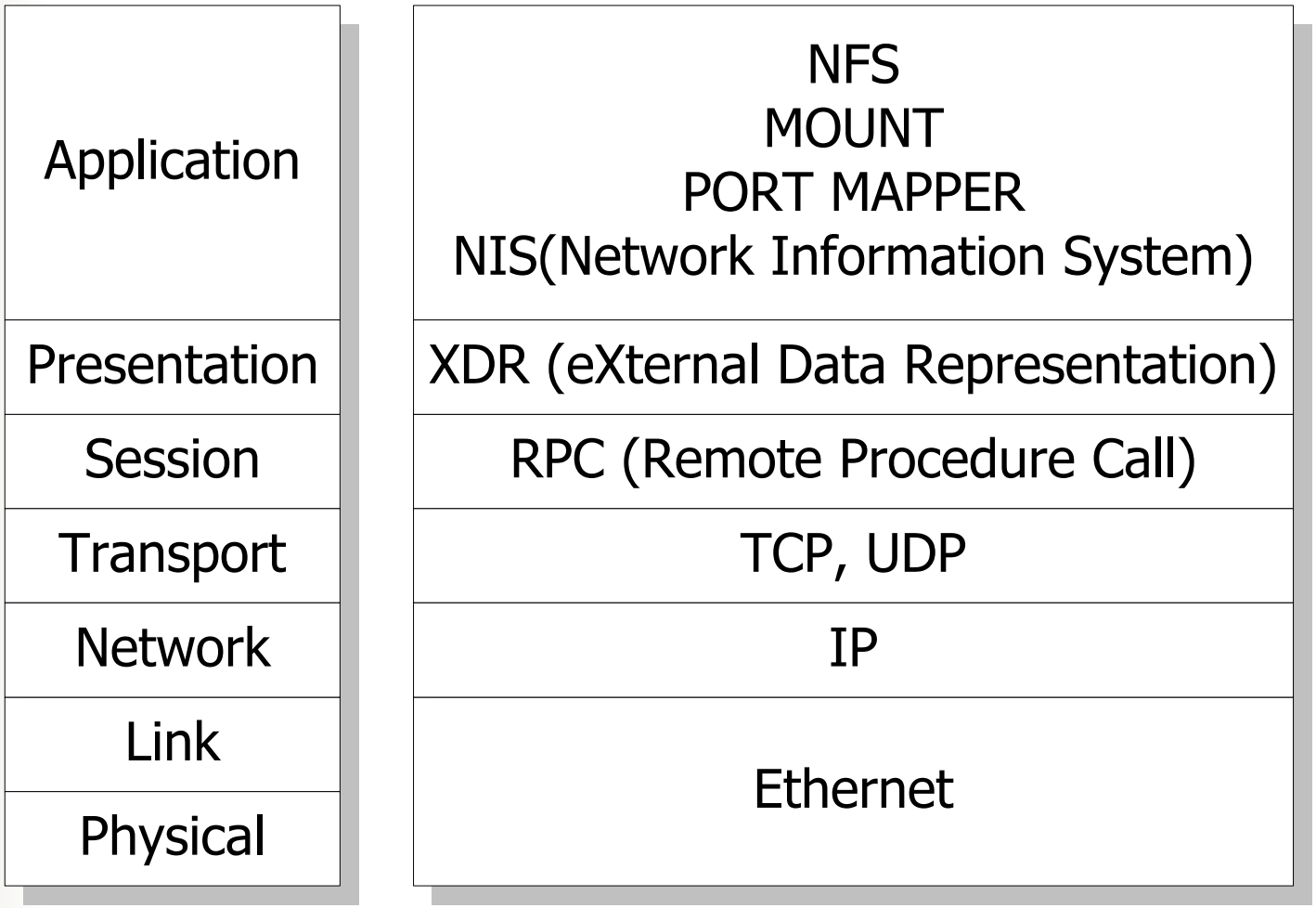
# Introduction to File Systems

- The most widely used on Linux is *ext2fs* (extended 2 file system)
- Every file is represented by an “inode”
  - A file descriptor holding, among other things, file access permissions, physical block addresses holding data, etc.
- The NFS server’s task is to give clients the inodes they want to access
- An NFS server gives an additional net layer allowing remote machines to handle the inodes

# The NFS Protocol

- NFS is built from 4 distinct protocols:
  - nfs
    - File creation, searching, reading, writing
    - Authentication and statistics
  - mountd
    - Mounting of “exported” systems for access via nfs
  - nsm
    - Network Status Monitor
    - Monitors a client or server machine’s status
  - nlm
    - Network Lock Manager
    - Avoid simultaneous data modification by multiple clients

# The NFS Protocol

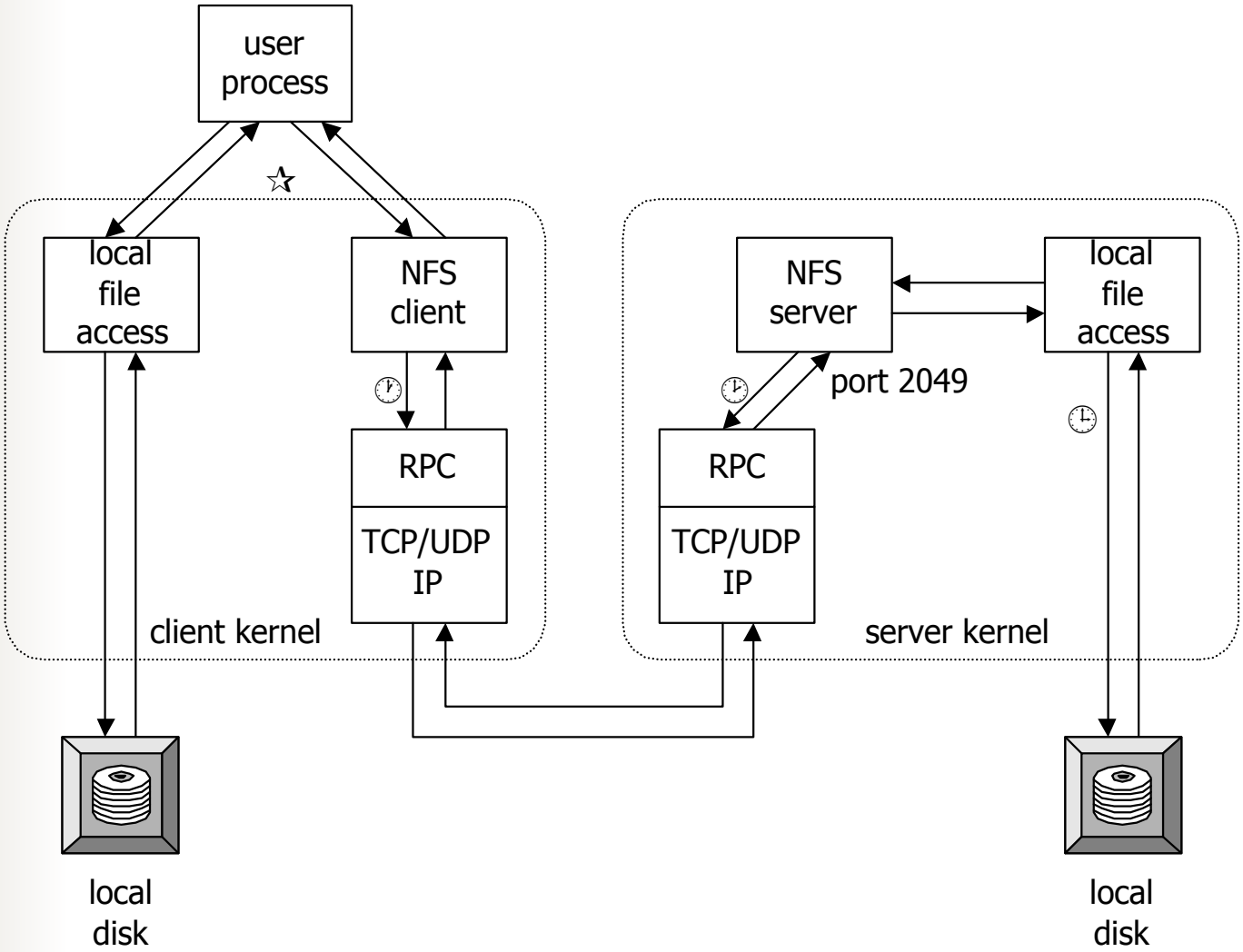



***OSI Model*** ***NFS Protocol Layers***

# The NFS Protocol

- There are two NFS Versions: nfsv2 & nfsv3
- The protocol revolves around the *filehandle*
  - A data structure allow unique identification of a file system object
  - Contains the file inode and an entry representing the device where the file resides
- View NFS as a file system embedded within a file system

# The NFS Protocol



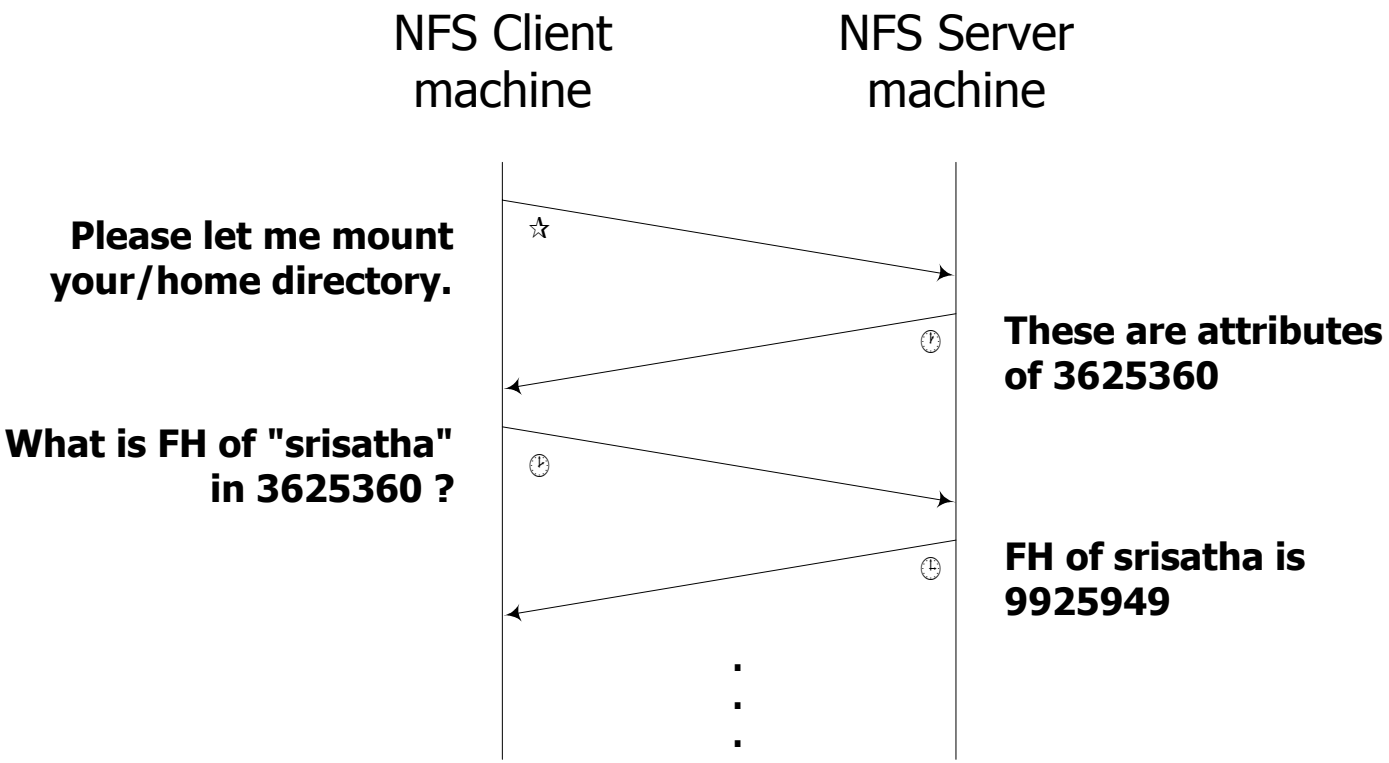
# File Handles

- How does a server know which file/directory the client needs to access?
  - At first, client obtains a file handle for root of the file system
  - File handle is opaque to the client
  - Client sends file handle to server when referencing a file/directory
  - No need to use the full path names
- “The file handle can contain whatever information the server needs to distinguish an individual file”

# File Handles

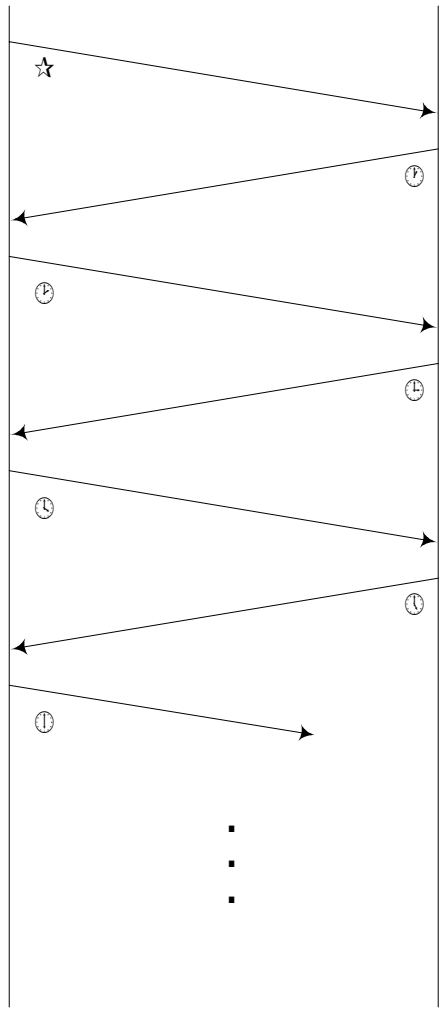
**Mount** : When NFS client starts up and mounts home directory

**NFS** : When a user login



# Example of File Handles

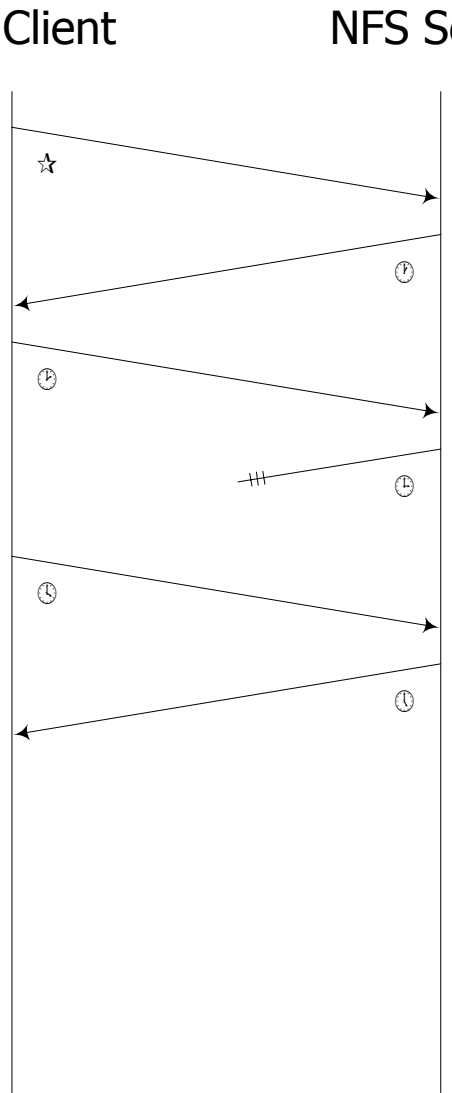
Suppose : (tcpdump-C) client needs to cat the file sub2/myname.txt under the current directory (~/srisatha)



# Statelessness

- What is statelessness ?
  - Server does not need to maintain protocol state about it's client
  - Server does not keep previous request information
  - Client keeps track of all information required to send requests to the server
- Advantage :
  - If server crashes, no state information lost
  - Client needs only retransmit a request until the server responds

# Why is idempotent important ?




Suppose : (tcpdump-D) client needs to remove the file sub2/myname.txt

# Idempotent procedures

- Can be executed more than once by the server and still return the same result
- Stateless protocol requires idempotent operation
- How to makes all NFS requests idempotent:
  - Server records recently performed operations in cache
  - Server checks in cache for duplicate requests
  - Server returns the previous result if it is a duplicate

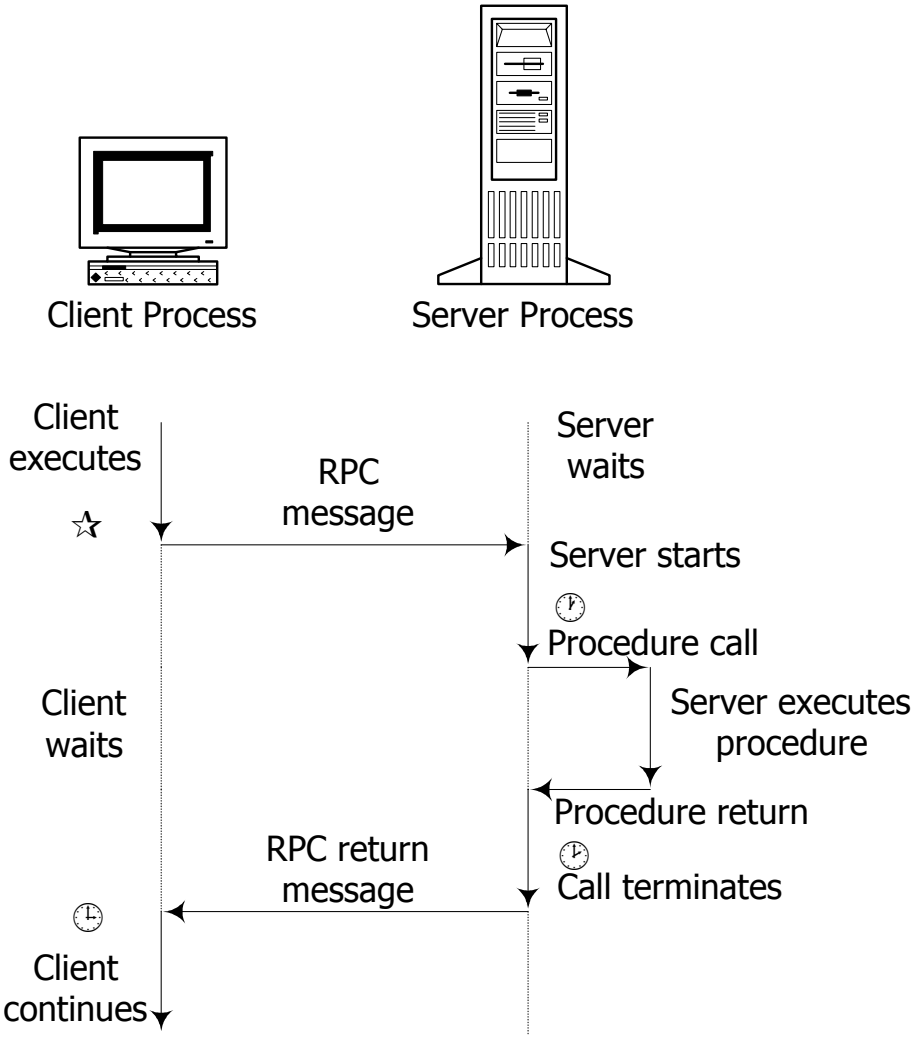
# Should NFS use TCP or UDP ?

- From the beginning, NFS used UDP
  - Most NFS systems were on LAN
  - High overhead if using TCP
- Currently, NFS across WAN needs TCP
  - Reliability and congestion control
  - Both sides set TCP's keep alive option
  - If server crashes, client opens new TCP connection
  - If client crashes, server will terminate the connection after the next keep alive probe

# The NFS Protocol

- Each relies on *Remote Procedure Calls* (RPC) and *Portmap* (also called rpc.portmap).
- An RPC server tells portmap which port will be used and the managed RPC number
- A client contacts portmap to get port number of desired server program
- RPC packets are addressed to the corresponding port
- Use the `rpcinfo -p` command to obtain details on services

# How does RPC works ?



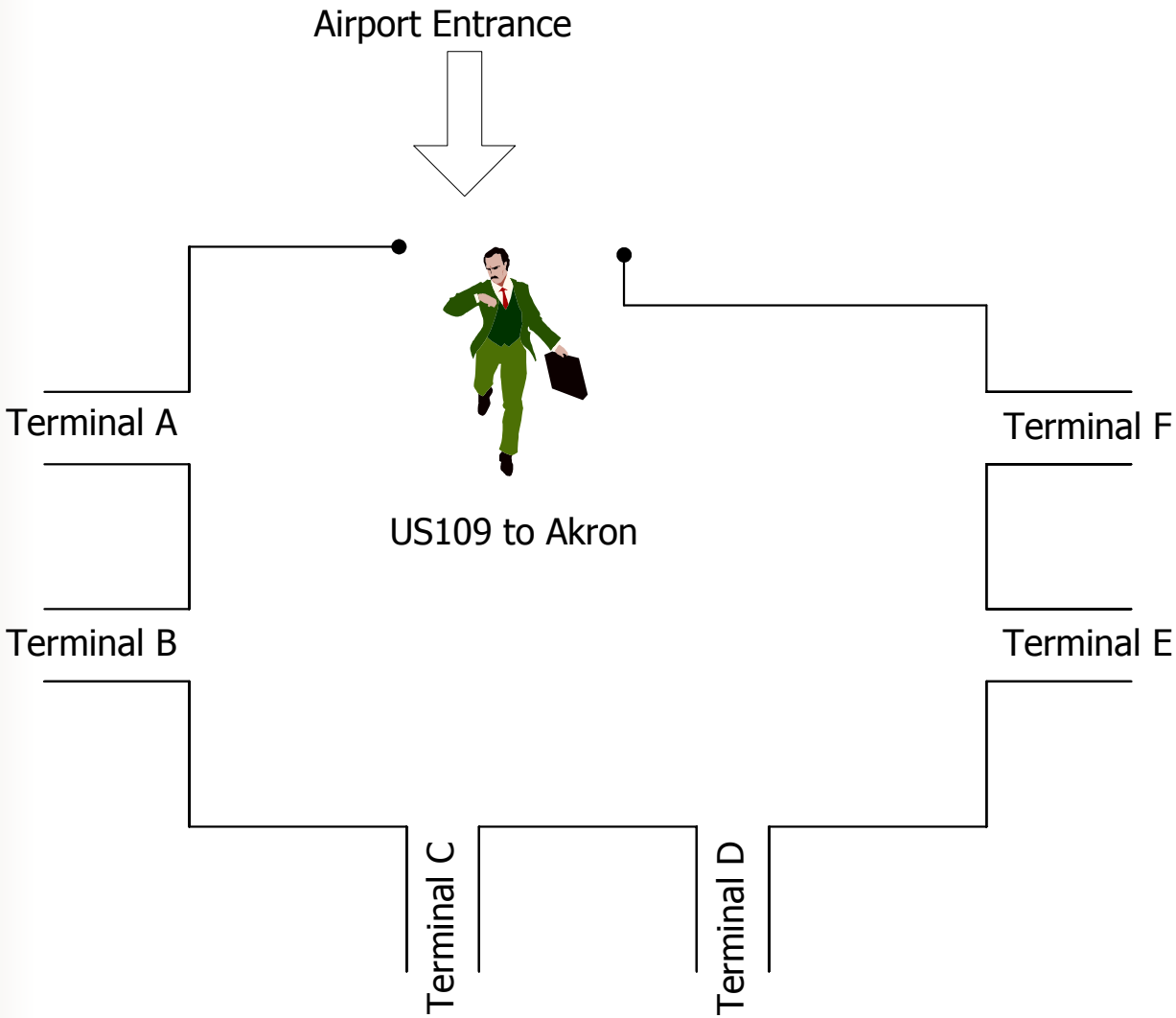
# RPC versus local procedure call

- Error handling:
  - failures of the server or network must be handled
- Global variables:
  - arguments cannot be passed as global variables
- Performance:
  - slower than local procedure calls
- Authentication:
  - RPC can be transported over insecure networks

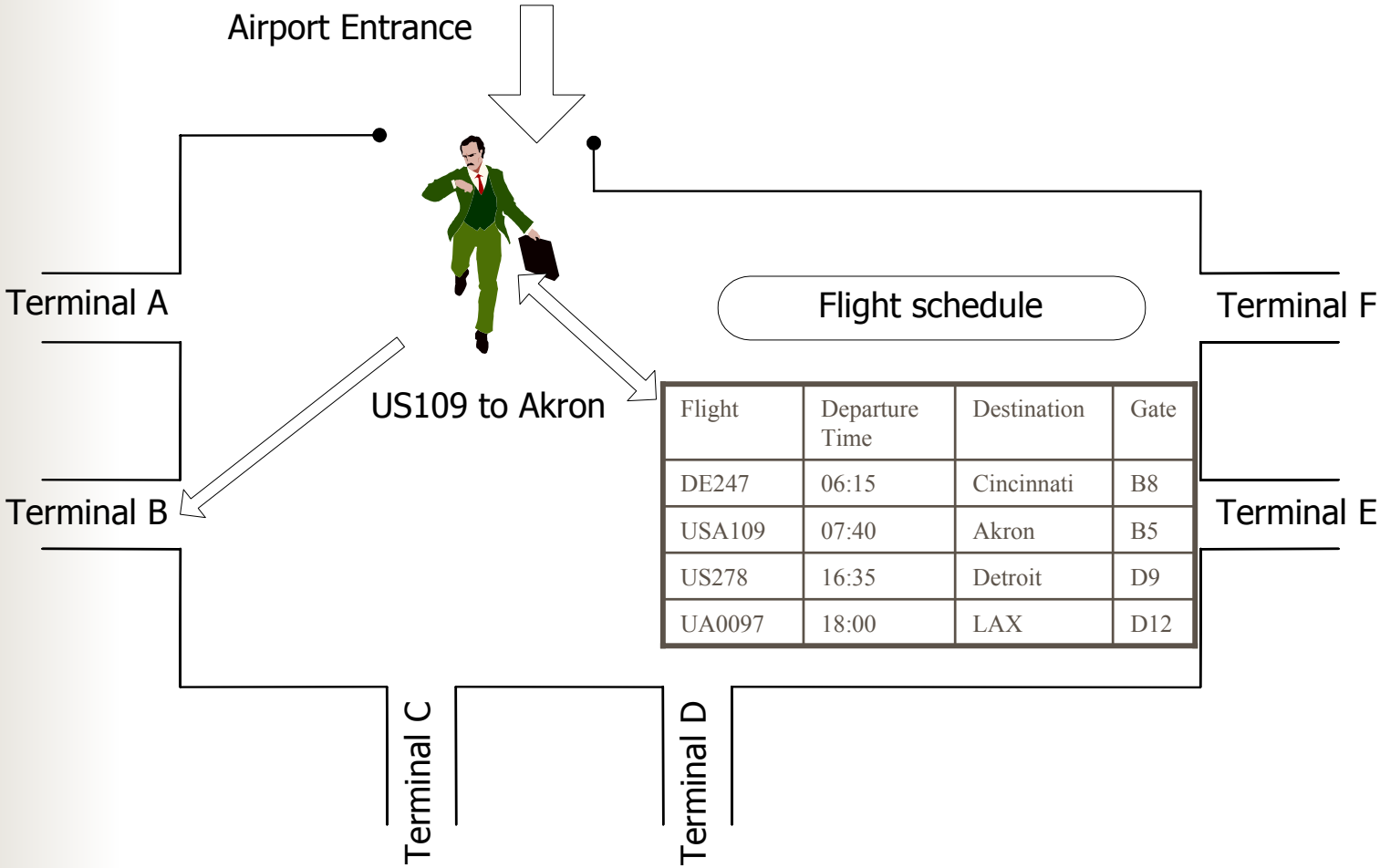
# List of RPC Programs

```
program vers proto   port  service
100000    4   tcp    111  portmapper
100000    3   tcp    111  portmapper
100000    2   tcp    111  portmapper
100000    4   udp    111  portmapper
100000    3   udp    111  portmapper
100000    2   udp    111  portmapper
100003    2   udp   2049  nfs
100003    3   udp   2049  nfs
100003    2   tcp   2049  nfs
100003    3   tcp   2049  nfs
100005    1   udp  32890  mountd
100005    2   udp  32890  mountd
100005    3   udp  32890  mountd
100005    1   tcp  32870  mountd
100005    2   tcp  32870  mountd
100005    3   tcp  32870  mountd
```

# Port Mapper : Analogy

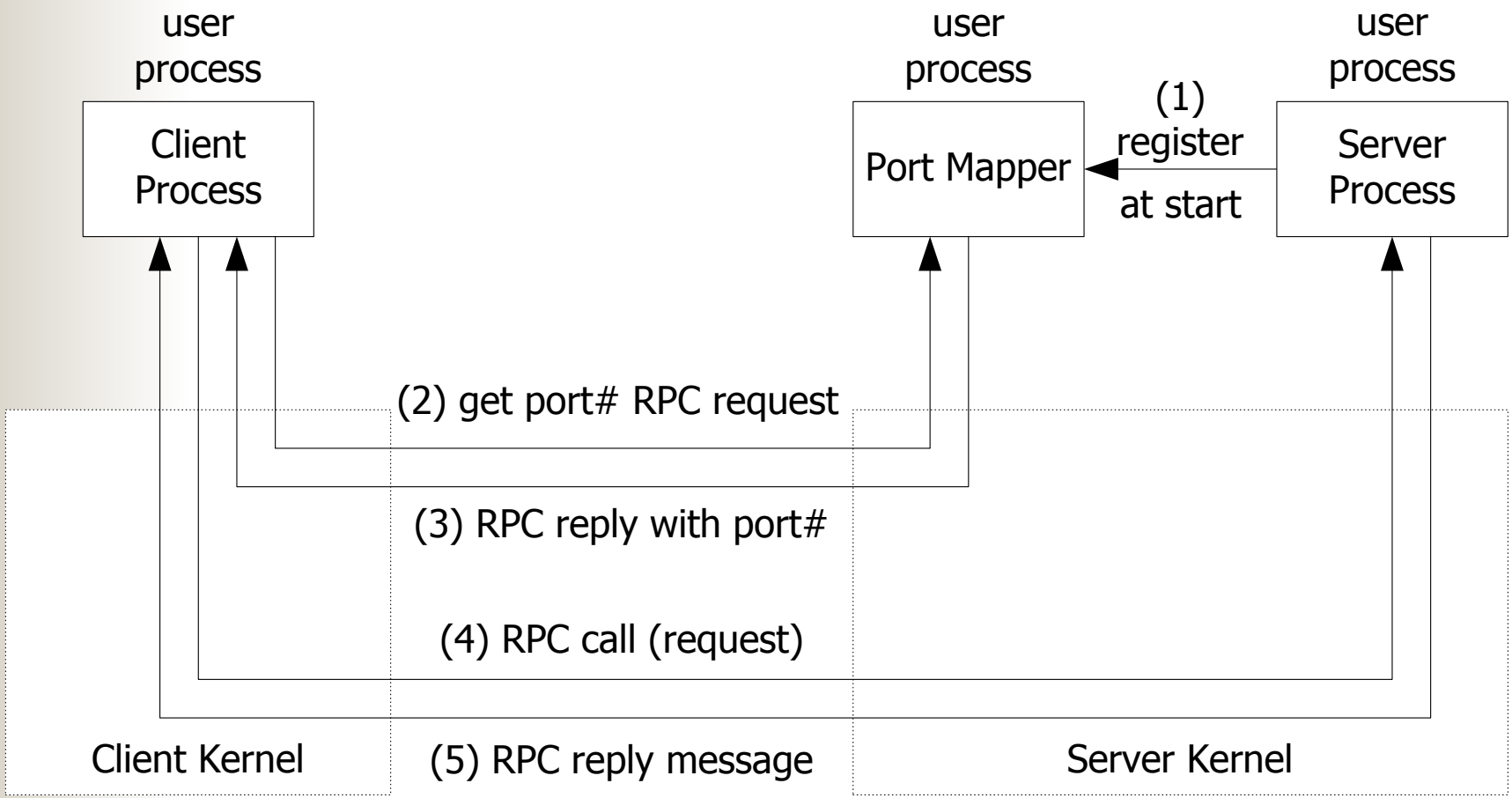


# Port Mapper : Analogy




| Flight | Departure Time | Destination | Gate |
|---|---|---|---|
| DE247 | 06:15 | Cincinnati | B8 |
| USA109 | 07:40 | Akron | B5 |
| US278 | 16:35 | Detroit | D9 |
| UA0097 | 18:00 | LAX | D12 |

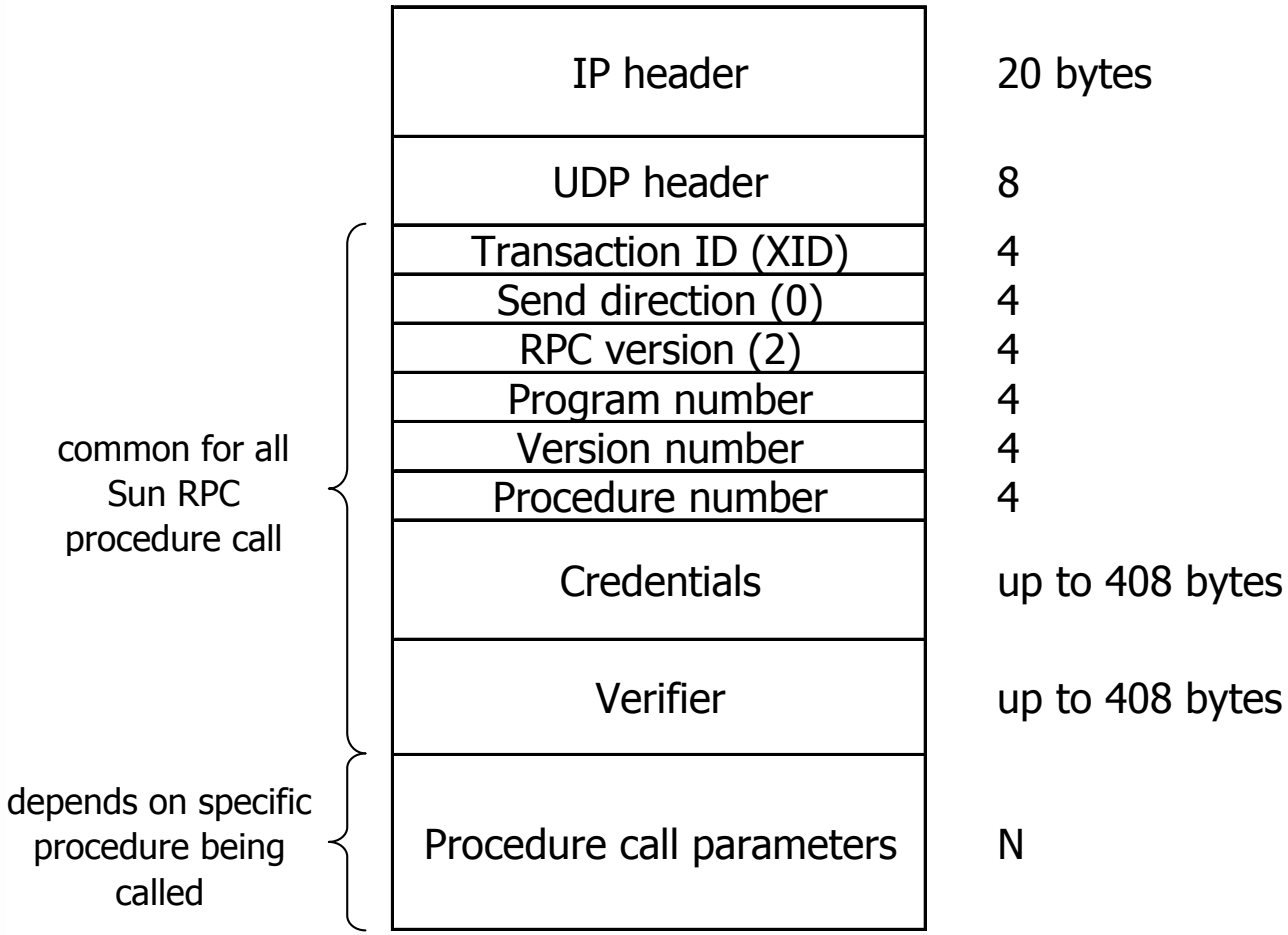
# Port Mapper/RPCBIND



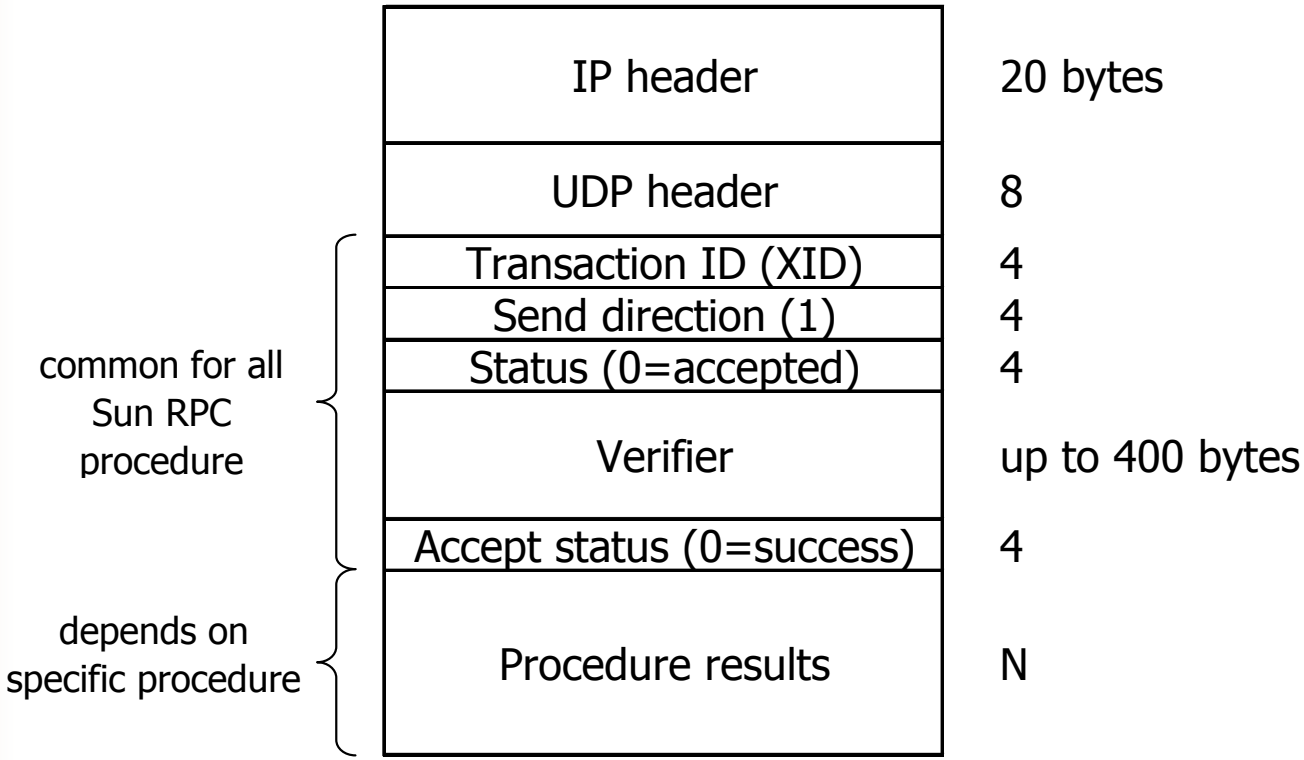
# NFS procedures

| ***NFS Procedures*** | ***Functions*** |
|---|---|
| LOOKUP | Returns a file handle and attribute corresponding to a file name in a specified directory. |
| MKDIR | Create a directory. |
| RMDIR | Delete a directory. |
| READDIR | Read a directory. Used by the Unix *ls* command, for example. |
| RENAME | Rename a file. |
| REMOVE | Delete a file. |
| CREATE | Create a file. |
| READ | Read from a file, by specify the file handle, starting offset and max. no. of bytes to read (up to 8192). |
| WRITE | Write to a file. |
| GETATTR | Returns the attributes of a file: type of file, permissions, size, owner, last-access time, and so on. |
| SETATTR | Set the attributes of a file: permissions, owner, group, size,and last-access and last-modification time. |
| LINK | Create a Unix hard link to a file. |
| SYMLINK | Create a symbolic link to a file. |
| READLINK | Returns the name of the file to whidh the symbolic link points. |
| STATFS | Returns the status of a file system. Used by the Unix *df* command, for example. |

# Format of RPC call



# Format of RPC reply



# NFS Server Configuration

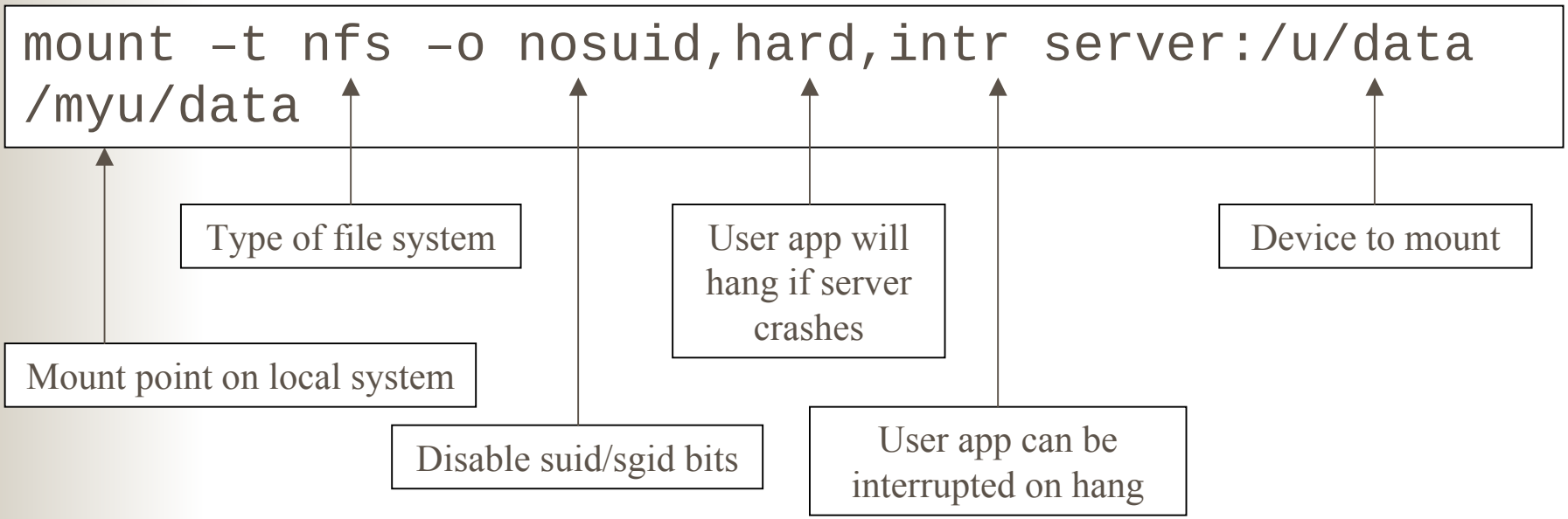
- One configuration file: `/etc/exports`
- Defines a location, a list of authorized clients, and options
- Client identified by
  - Machine name
  - Wildcards on a domain name
  - A *netgroup* (if NIS is used)
  - An IP address
- Options include: `rw`, `ro`, `root_squash`, `all_squash`, `anonuid`

# NFS Server Starting

- Start `rpc.mountd` and `rpc.nfs`
- Optionally start `rpc.statd` and `rpc.lockd`
- Use `rpcinfo -p` to check if working
- Files:
  - `/var/lib/nfs/rmtab` contains client information
  - `/var/lib/nfs/etab` contains detailed export information
  - `/proc/fs/nfs/exports` contains list of clients
  - `/var/lib/nfs/xtab` contains explicit machine names
- If `/etc/export` is updated use exportfs command to inform servers

# Client Access

- Use mount command:




- Update `/etc/fstab` to have device mounted at startup

# Client Access

```
/dev/dasda1     swap                        swap            defaults   0   0
/dev/dasdb1     /                           ext2            defaults   1   1
/dev/dasdc1     /home                       ext2            defaults   1   2
/dev/dasdd1     /MDISK/0200                 ext2            defaults   1   3
/dev/dasde1     /MDISK/0290                 ext2            defaults   1   4
/dev/dasdf1     /MDISK/0210                 ext2            defaults   1   5

alf7.software-ag.de:/FS/fs0820              /FS/fs0820 nfs
intr,rsize=1024,wsize=1024,soft,bg,retry=1000
daeux42:/vol/vol2/fs1028                   /FS/fs1028 nfs
intr,rsize=1024,wsize=1024,soft,bg,retry=1000
daeux40:/vol/vol0/fs1005                   /FS/fs1005 nfs
intr,rsize=1024,wsize=1024,soft,bg,retry=1000
daeux40:/vol/vol0/fs1021                   /FS/fs1021 nfs
intr,rsize=1024,wsize=1024,soft,bg,retry=1000,vers=2
```

# Security Considerations

- Implicit trusted relationship between server and client
- Client cannot blindly trust server
  - Use the `nosuid` option on mount command
  - May use `noexec` option to forbid execution
- Server cannot blindly trust client
  - Use the `root_squash` to map root to nobody
  - `anonuid` and `anongid` can be used to change map
  - Forbid default access using `/etc/hosts.deny`
- Establish good firewall rules

# nfsv2 v nfsv3

- The file handle in v2 is a fixed-size array of 32 bytes. With v3 it becomes a variable-length array up to 64 bytes. A variable-length in XDR is encoded with a 4-byte count, followed by the actual bytes. This reduces the size of the file handle on implementations such as UNIX that only need about 12 bytes, but allows non-UNIX implementations to maintain additional information.

# nfsv2 v nfsv3

- v2 limits the number of bytes per READ or WRITE RPC to 8192 bytes. This limit is removed in v3, meaning an implementation over UDP is limited only by the IP datagram size (65535 bytes). This allows larger read and write packets on faster networks.

# nfsv2 v nfsv3

- File sizes and the starting byte offsets for the READ and WRITE procedures are extended from 32 to 64 bits, allowing larger file sizes.
- A file's attributes are returned on every call that affects the attributes. This reduces the number of GETATTR calls required by the client.

# nfsv2 v nfsv3

- WRITEs can be asynchronous, instead of the synchronous WRITEs required by v2. This can improve WRITE performance.

# nfsv2 v nfsv3

- One procedure was deleted (STATFS) and seven were added:
  - ACCESS check file access permissions
  - MKNOD create a UNIX special file
  - READDIRPLUS returns names of files in a directory along with their attributes
  - FSINFO returns the static information about a filesystem
  - FSSTAT returns the dynamic information about the filesystem
  - PATHCONF returns the POSIX.1 information about a file
  - COMMIT commit previous asynchronous writes to stable storage

# Version 2 & Version 3 Mixes

"When an NFS connection is made to a drive, the NFS client tries to talk using the [v3] protocol (it asks the NFS server for a response to the NULL [v3] procedure). If that fails, then [v2] is all that will be used. If is succeeds, then we assume that [v3] is available, and we will use the [v3] async-write procedure and the [v3] readdirplus procedure. If either of these procedures generates an OPNOTSUP failure during normal operation[,] we fall back to [v2]. All other ancillary operations (file locking, create, remove, etc.) are done using [v2], since we need a v2 path anyway and the operations are the same. It is important to realize that [v3] has some features which are a tremendous boon (readdirplus in particular speeds things up tremendously) and other features which are interesting for other reasons ([v3] locks support 64-bit offsets and lengths, but this is not particularly interesting for NT which needs only 32-bit values)."