EEL-4736/5737 Principles of Computer System Design

Lecture Slides 7

Textbook Chapter 4

Case study – Network File System

Introduction

- We have seen a case study of a the UNIX file system
  - Within a computer
- Let us now overview a widely used client-service system that supports the file system abstraction across multiple computers

Distributed file systems

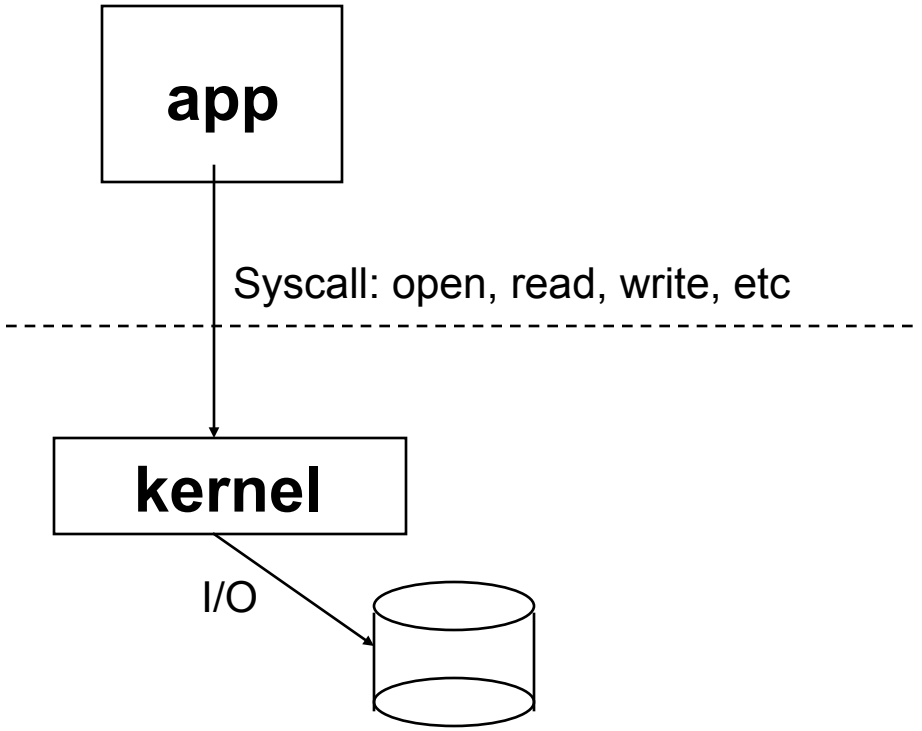
- Maintain the abstraction of a file system, while allowing access to remote I/O resources over networks
- Examples:
  - NFS (Network File System):
    - Common in most Unix environments; targets local-area networks
  - SMB (Server Message Block):
    - Common in Windows environments
  - AFS (Andrew File System):
    - Targets wide-area networks; more difficult to setup and administer, less common

Example - NFS

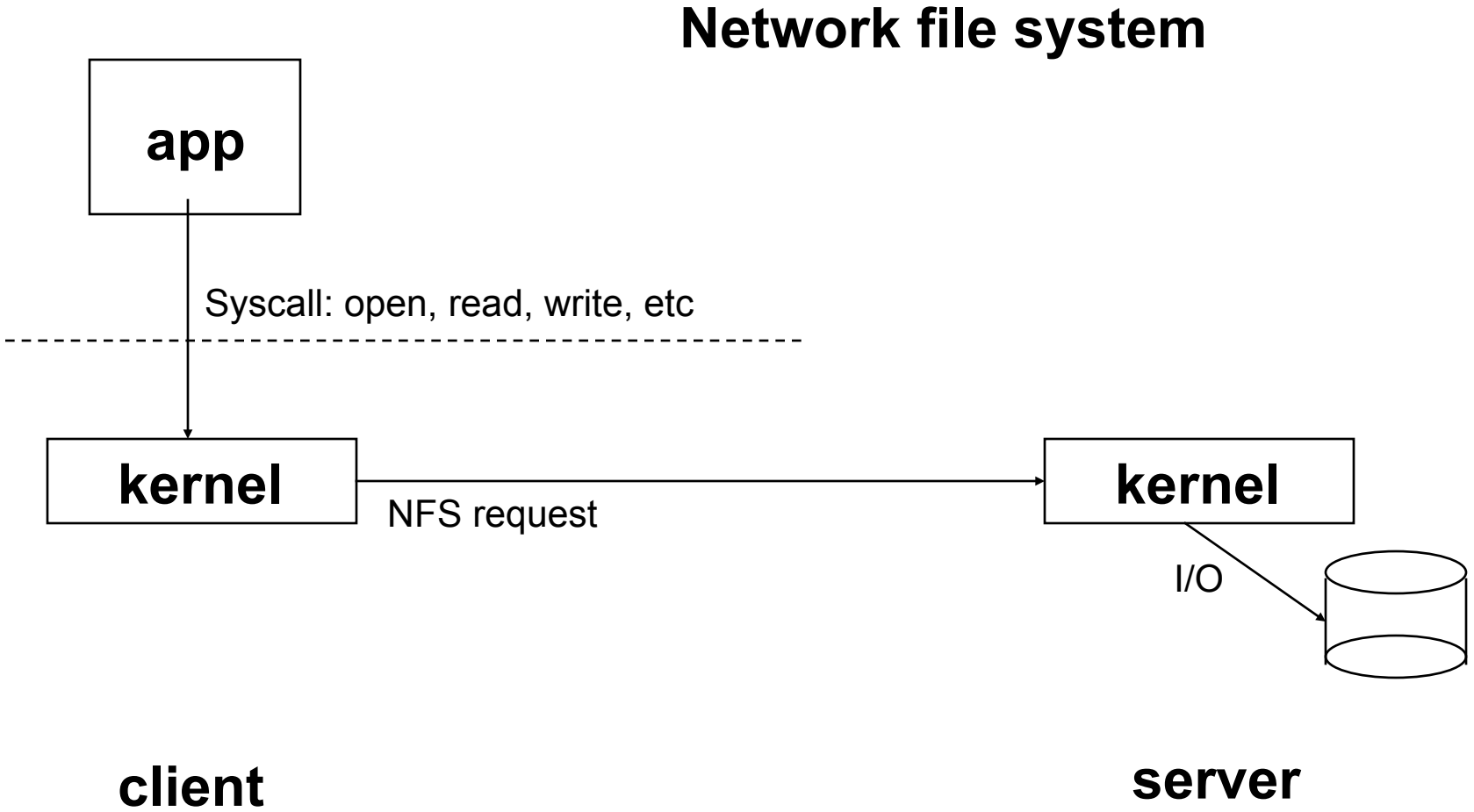
- Distributed file system protocol initially developed by Sun
  - Versions 2, 3, 4
- Unix-oriented
- Goal:
  - support a distributed file system (with same semantics as local Unix file systems) across heterogeneous clients/servers on a TCP/IP local area network

User's perspective

Local file system



User's perspective



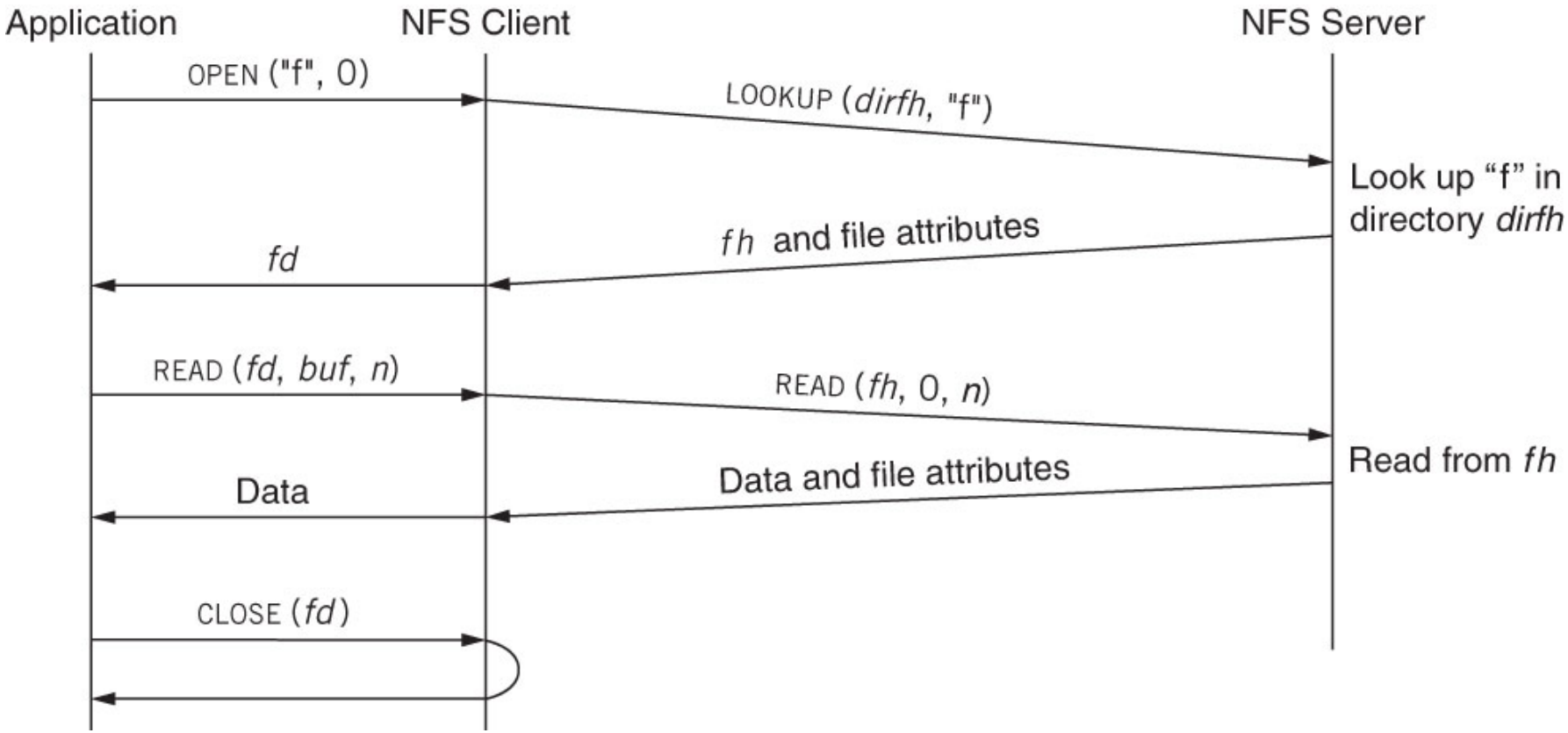
NFS implementation

- Previous picture would suggest a method of sending system calls “over the network”
  - Not how NFS operates
  - System calls differ among O/S families, and NFS was designed to support heterogeneous nodes

NFS implementation

- Rather:
  - NFS defines a protocol that is independent from an O/S's system call definition
    - Try to match closely typical system calls for efficiency
  - NFS uses remote procedure calls (RPC) to issue protocol requests over the network
    - RPC also designed with portability across O/Ss in mind

Example interaction

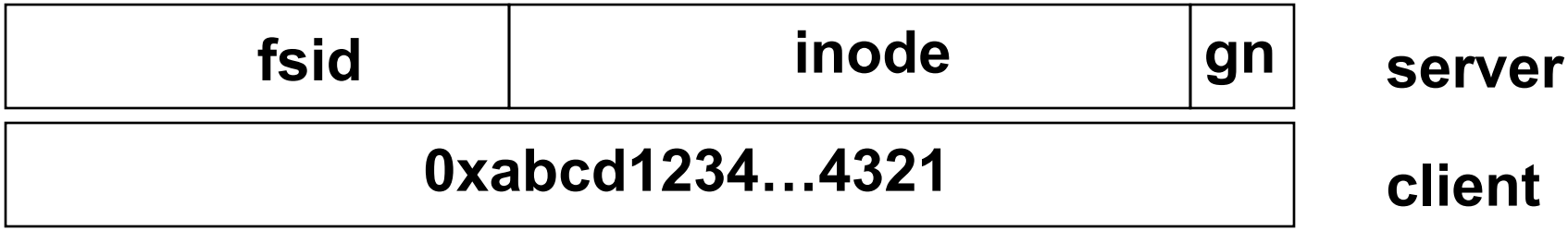


The file handle

- An identifier used by the client to refer to files and directories stored in the server
  - NFS-V2: 32-byte data
  - NFS-V3: 64 bytes
- File operations use file handles as arguments
  - LOOKUP returns file handles based on a file's name and the file handle of its directory
    - MOUNT protocol returns the root file handle

The file handle

- It's an "opaque" name
  - Server uses information structured in file handle to locate an object
    - fsid – which file system it belongs?
    - Inode number – at the server's file system
    - Generation number
  - Client does not 'peek into' file handle
    - Rather, use it entirely as a name



The file handle

- Similar to an inode number from the client's perspective
  - Used by client in all references to objects
- Why not path names?
  - Performance argument – faster lookup
  - UNIX semantics - if one client renames a directory holding a file opened by another client, reference by name would point the open file to wrong object
- Why generation number?
  - If an open file is unlinked and recreated at another client, prevents the file handle from being associated with the new file

File handles vs path names

Client 1:

chdir(“dir1”)

fd <- open(“f”,RO)

read(fd,buf,n)

Client 2:

rename(“dir1”,”dir2”)

rename(“dir3”,”dir1”)

READ(“dir1/f”)? Read dir2/f – incorrect

READ(fh) – inode, not path name

Generation count

Client 1:

fd <- open(“f”,RO)

read(fd,buf,n)

Client 2:

unlink(“f”)

fd<-open(“f”,CREAT)

Ideally, in UNIX semantics:

client1’s read reads from open file

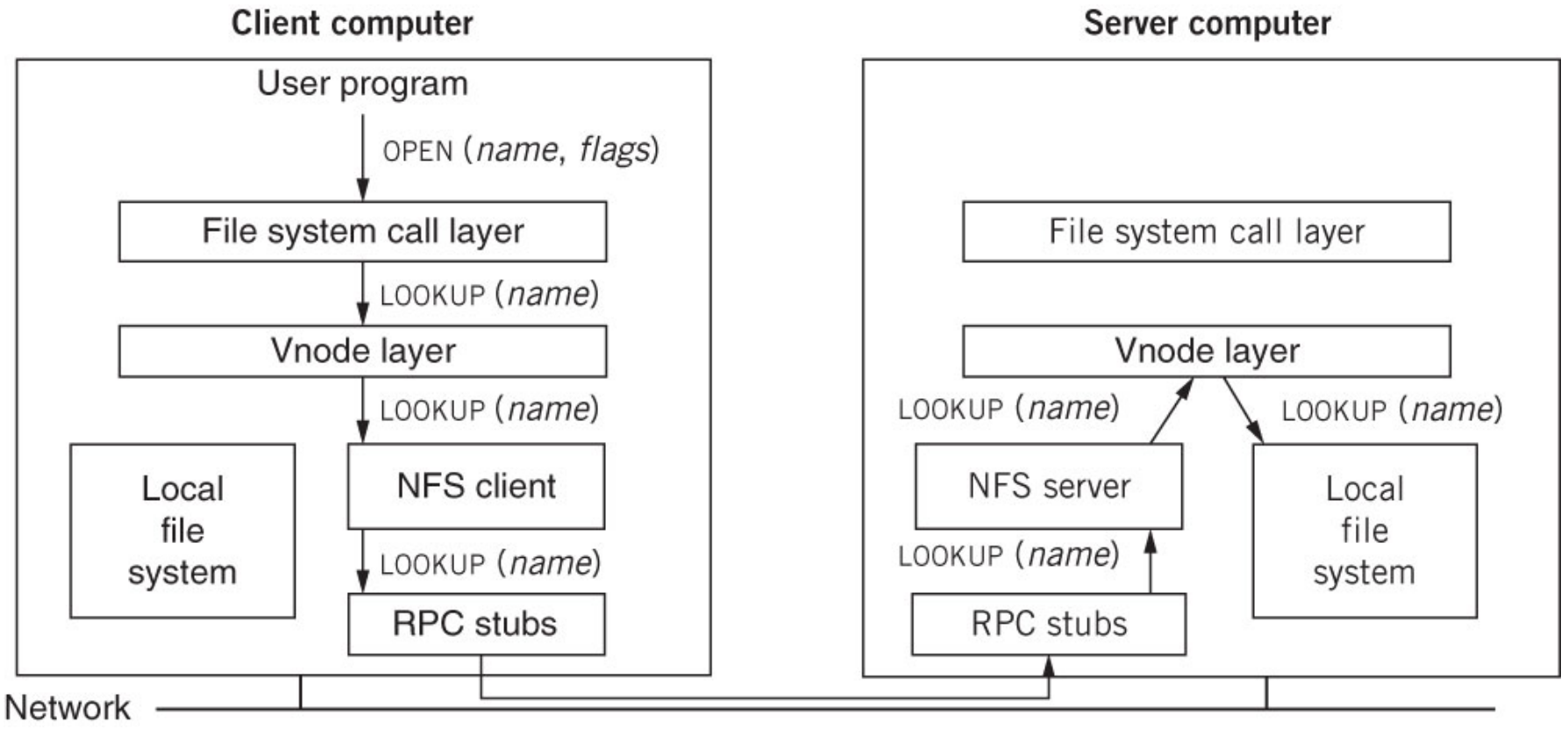
Unacceptable behavior:

client1’2 read reads from new file

The virtual node layer

- Abstracts whether a file or directory is in a local file system or remote file system
  - Layer of indirection between system call and data stored on disk inodes or on file system client's data structure
  - Additional advantage: can be used to support various kinds of local file systems

NFS implementation



Example: NFS-V2 protocol

- http://www.faqs.org/rfcs/rfc1094.html
- Protocol defines 18 RPC calls
- E.g. READ, WRITE, REMOVE, RENAME
  - Close matches to read(), write(), unlink(), rename()
- E.g. LOOKUP
  - No close match; used to lookup file names and associate them with file “handles” to be used between client/server
  - Similar to the Lookup primitive we studied

Example - Lookup

- Argument: diropargs data structure
- Return: diropres data structure

```
struct diropargs {
    fhandle dir;              // directory to look up file
    filename name; };         // file name
```

Example - Lookup

Return:

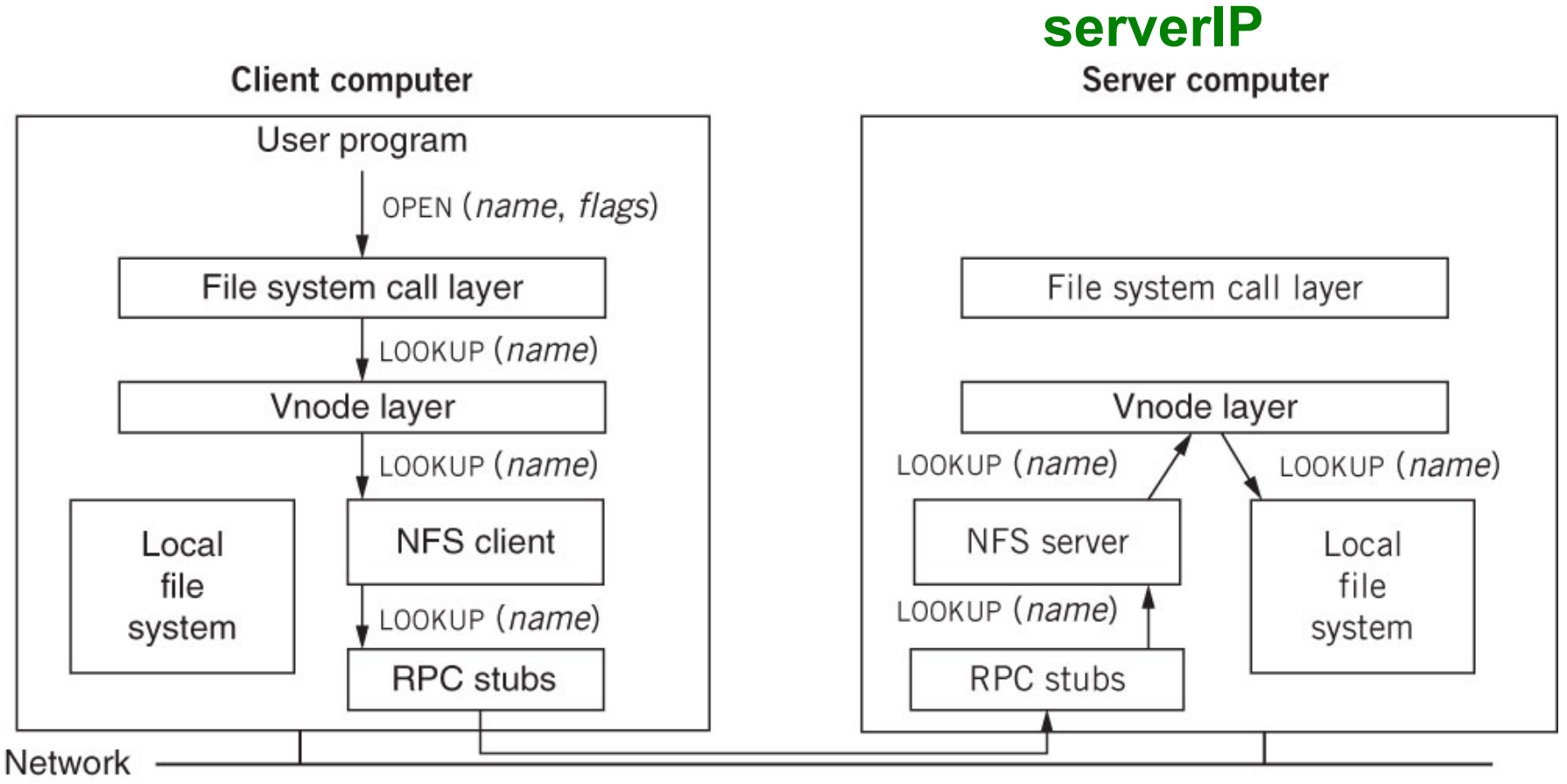
```
union diropres switch (stat status) {
    case NFS_OK: struct {
       fhandle file;
       fattr attributes; } diropok;
    default: void;
};
```

Attributes

- struct fattr {
  ftype type;
  unsigned int mode;
  unsigned int nlink;
  unsigned int uid;
  unsigned int gid;
  unsigned int size;
  unsigned int blocksize;
  unsigned int rdev;
  unsigned int blocks;
  unsigned int fsid;
  unsigned int fileid;
  timeval atime; timeval mtime; timeval ctime;

};

Example



fs1: /mnt/nfs

fs1: /export/data/file.txt
fs2: /local/sys/…

MOUNT (serverIP, "/export", "/mnt/nfs")

Binding RPCs

- RPC server identified by:
  - Address of server machine
  - Port in which RPC listens to request
  - Program identifier
  - Version identifier
- Example: NFS
  - Server = NFS server's IP
  - Port = 2049 (typically)
  - Identifier = 100003
  - Version = 2 or 3

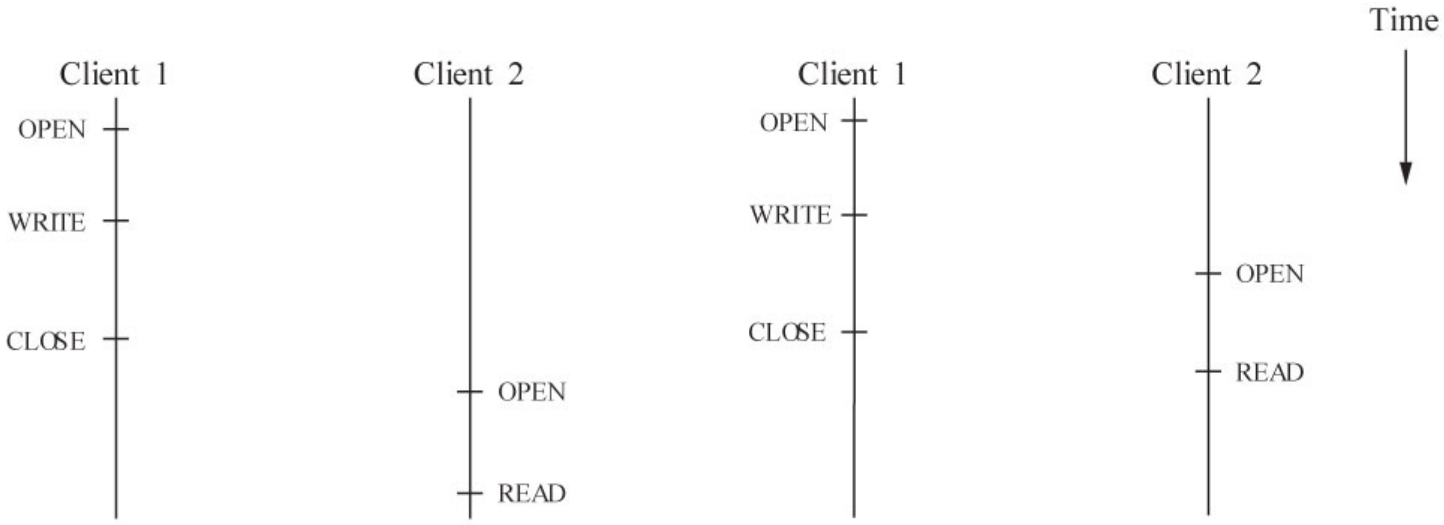
Coherence

- Performance enhancement:
  - Cache read data on NFS client's memory
    - Speed up later reads – cache hits
  - Buffer writes on NFS client's memory
    - Faster completion of requests (no need to wait for RPC acknowledgment)
    - Coalesce write requests
- Challenge – what happens when there are multiple clients?
  - Local file system: read observes data from latest write. Not the case now

Close-to-open consistency

- On an open, send GETATTR RPC and check client vnode's modification time against time on the server side
  - If server modification time is later, discard cached blocks for the file
- On a close, send WRITEs to all blocks that may still remain on client's cache
- Pragmatic approach that works if clients don't open files concurrently

Close-to-open consistency



The road ahead

- For more depth in this topic, you should consider taking Distributed Computing (EEL-6935)
- Next topic – enforcing modularity with virtualization
  - Sections 5.1, 5.2