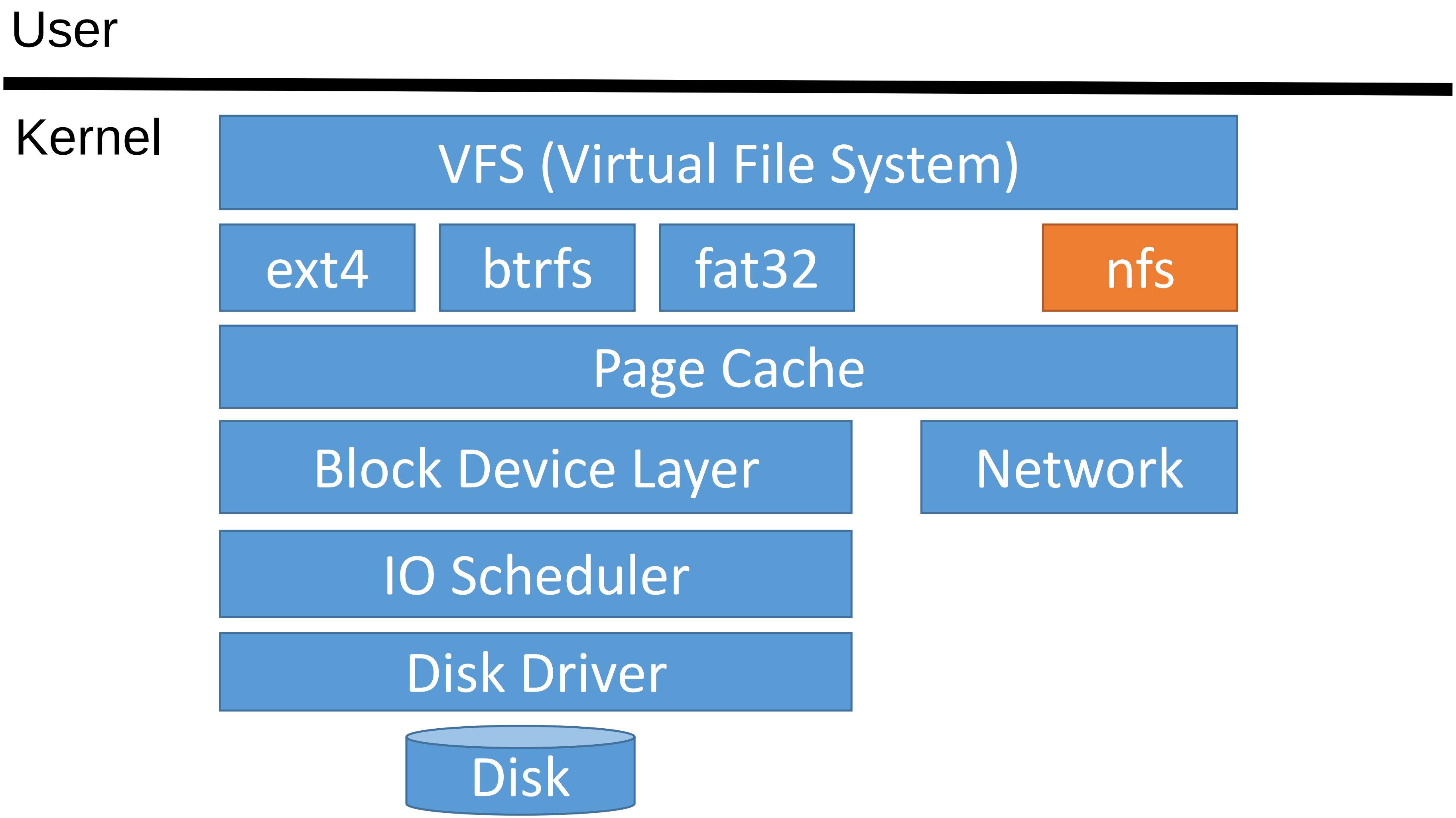
Network File System (NFS)

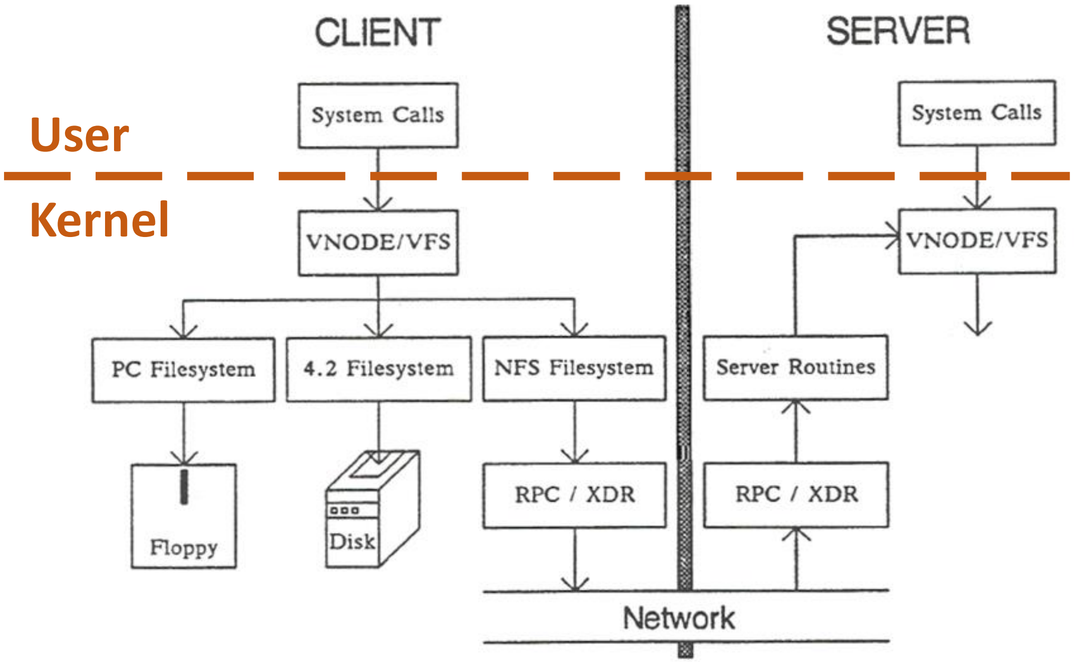
Nima Honarmand

A Typical Storage Stack (Linux)



NFS Idea

- A client/server system to share the content of a file system over network
- Translate VFS requests into **Remote Procedure Calls (RPC)** to server
  - Instead of translating them into disk accesses



Source: Sandberg et al., 1985

Remote Procedure Call (1)

- Intuition: create wrappers so calling a function on another machine feels just like calling a local function

Machine A

```
int main(…) {
  int x = foo(”hello”);
}

int foo(char *msg) {
  send msg to B
  recv msg from B
}
```

Machine B

```
int foo(char *msg) {
  …
}

void foo_listener() {
  while(1) {
    recv, call foo
    send result to B
  }
}
```

What it feels like for programmer

Remote Procedure Call (2)

- Intuition: create wrappers so calling a function on another machine feels just like calling a local function

Machine A

```
int main(…) {
  int x = foo("hello");
}

int foo(char *msg) {
  send msg to B
  recv msg from B
}
```

(1) (2) (3) (4)

Machine B

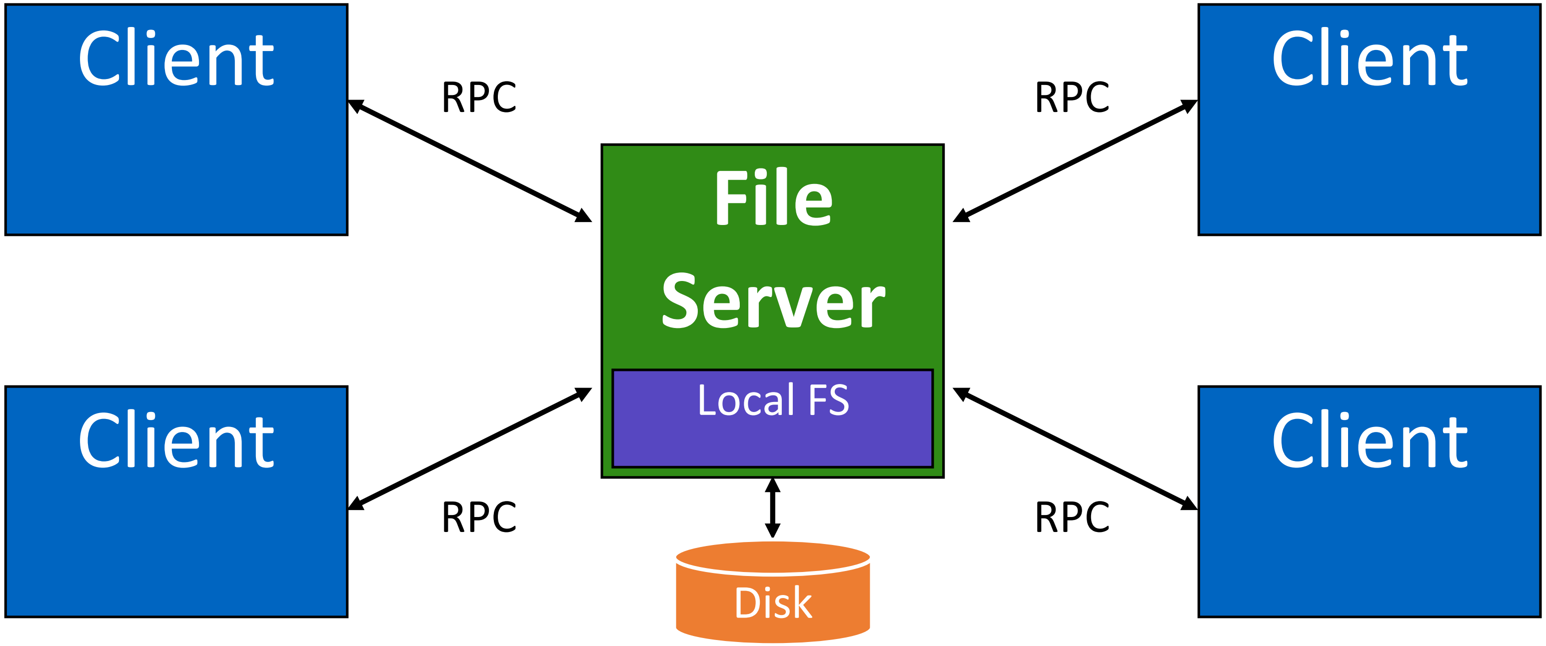
```
int foo(char *msg) {
  …
}

void foo_listener() {
  while(1) {
    recv, call foo
    send result to B
  }
}
```

Actual Calls

Remote Procedure Call (3)

- There is a pre-assigned procedure ID for each remote call
- Client side:
  1) Pack procedure ID and all its arguments in an RPC request packet (aka. ***serialization*** or ***marshalling***)
  2) Send the request to the server
  3) Wait for the response
  4) unpack results (aka. ***deserialization*** or ***unmarshalling***) & return to caller
- Server side:
  1) Wait for and receive the request packet
  2) Deserialize the request content (procedure ID and arguments) into appropriate data structures
  3) Service the request
  4) Serialize results into an RPC response packet and send it to the client

General NFS Architecture



- Server exports the NFS volume
  - Basically assigns a port number to it
- Each client “mounts” the NFS volume somewhere in its directory tree

Challenges of NFS Protocol Design

- Both server or client can crash (i.e., lose state)
- Server and client can be temporarily disconnected (e.g., lost or corrupted packets)
- How to coordinate multiple clients actions?
  - Client-side cashing
  - inode reuse
- Buffering writes in the server
- …

Protocol Design: First Attempt

- Attempt: Wrap regular UNIX system calls using RPC

- open() on client calls open() on server
- open() on server returns fd back to client

- read(fd) on client calls read(fd) on server
- read(fd) on server returns data back to client

Challenge 1: Dealing w/ Crashes

- What about crashes?

```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);
…
read(fd, buf, MAX);
```

Server crash!

nice if acts like a slow read

- Imagine server crashes and reboots during reads…

Stateful vs. Stateless Protocols (1)

- ***Stateful protocol***: server keeps track of past requests and client states
  - i.e., state persist across requests on the server
  - For example, keep track of open files and their cursor by each client
- ***Stateless protocol***: server does not keep track of past requests
  - Client should send all necessary state with a single request
  - E.g., server does not keep track of a client's open file cursor

Stateful vs. Stateless Protocols (2)

- Challenge of stateful: **recovery from crash**
  - Server side challenges:
    - Knowing when a client has crashed
    - Tracking state that needs to be cleaned up on such a crash
  - Client side challenges:
    - If server thinks we failed, must recreate server state
    - If server crashes and restarts, must recreate server state
- Drawbacks of stateless:
  - May introduce more complicated messages
  - And more messages in general

NFS is Stateless

- Every request sends all needed info
  - User credentials (for security checking)
  - File handle and offset
- Each request matches a VFS operation
  - NFSPROC_GETATTR, NFSPROC_SETATTR, NFSPROC_LOOKUP, NFSPROC_READ, NFSPROC_WRITE, NFSPROC_CREATE, NFSPROC_REMOVE, NFSPROC_MKDIR
  - There is no OPEN or CLOSE among NFS operations
    - That would make the protocol stateful
- Most requests need to specify a file
  - NFS ***file handle*** maps to a 3-tuple: (*server-fs*, *server-inode*, *generation-number*)

Challenge 2: Request Timeouts (1)

- Request sent to NFS server, no response received
  1) Did the message get lost in the network (UDP)?
  2) Did the server die?
  3) Is the server slow?
  4) Is the response lost or in transit?
- Client has to retry after a timeout
  - Okay if (1) or (2)
  - Potentially doing things twice if (3) or (4)
- But client can't distinguish between these cases!

  → Should make retries safe

Challenge 2: Request Timeouts (2)

- Idea: Make all requests ***idempotent***
  - <u>Requests should have same effect when executed multiple times</u>
    - Ex: NFSPROC_WRITE has an explicit offset, same effect if done twice
- Some requests not easy to make idempotent
  - E.g., deleting a file, making a directory, etc.
  - Partial remedy: server keeps a cache of recent requests and ignores duplicates

Challenge 3: inode Reuse

- Process A opens file ‘foo’
  - Maps to inode 30
- Process B unlinks file ‘foo’
  - On client, OS holds reference to the client inode alive
  - NFS is stateless, server doesn’t know about open handle
    - The file can be deleted and the server inode reused
    - Next request for inode 30 will go to the wrong file
- Idea: ***generation number*** as part of file handle
  - If server inode is recycled, generation number is incremented
  - Enables detecting attempts to access an old inode

Challenge 4: Client-Side Caching

- Client-side caching is necessary for high-performance
  - Otherwise, for every user FS operation, we'll have to go to the server (perhaps multiple times)
- Can cause consistency issues when there are multiple copies of data

Example:

- Clients **A** and **B** have file in their page cache
- Client **A** writes to the file
  - Data stays in **A**'s cache
  - Eventually flushed to the server
- Client **B** reads the file
  - Does **B** see the old content or the new stuff?
  - Who tells **B** that the cache is stale?
    - Server could tell, but only after **A** actually wrote/flushed the data
    - But this would make the protocol stateful — bad idea!

Consistency/Performance Tradeoff

- Performance: cache always, write when convenient
  - Other clients can see old data, or make conflicting updates
- Consistency: write everything to server immediately
  - And tell everyone who may have it cached
    - Requires server to know the clients which cache the file (stateful)
  - Much more network traffic, lower performance
  - Not good for the common case: accessing an unshared file

Compromise: Close-to-Open Consistency

- NFS Model: Close-to-Open consistency
- On `close()`, flush all writes to the server
- On `open()`, ask the server for the current timestamp to check the cached version's timestamp
  - If stale, invalidate the cache
  - Makes sure you get the latest version on the server when opening a file

Challenge 5: Removal of Open Files

- Recall: Unix allows accessing deleted files if still open
  - Reference in in-memory inode prevents cleanup
  - Applications expect this behavior; how to deal with it in NFS?
- On client, check if file is open before removing it
  - If yes, rename file instead of deleting it
    - `.nfs*` files in modern NFS
  - When file is closed, delete temp file
    - If client crashes, garbage file is left over ☹
  - Only works if the same client opens and then removes file

Challenge 6: Time Synchronization

- Each CPU's clock ticks at slightly different rates
  - These clocks can drift over time
- Tools like 'make' use file timestamps
  - Clock drift can cause programs to misbehave

```
make[2]: warning: Clock skew detected.
Your build may be incomplete.
```

- Systems using NFS must have clocks synchronized
  - Using external protocol like **Network Time Protocol** (NTP)
    - Synchronization depends on unknown communication delay
    - Very complex protocol but works pretty well in practice

Challenge 7: Security

- Local UID/GID passed as part of the call
  - UIDs must match across systems
  - Yellow pages (yp) service; evolved to NIS
  - Replaced with LDAP or Active Directory
- Problem with “root” (User ID 0) : root on one machine becomes root everywhere
- Solution: root squashing – root (UID 0) mapped to “nobody”
  - Ineffective security
    - Malicious client, can send any UID in the NFS packet

NFS Evolution

- The simple protocol was version 2 (1989)
- Version 3 (1995):
  - 64-bit file sizes and offsets (large file support)
  - Bundle attributes with other requests to eliminate stat()
  - Other optimizations
  - Still widely used today

NFSv4 (2000, 2003, 2015)

- Attempts to address many of the problems of v3
  - Security (eliminate homogeneous UID assumptions)
  - Performance
- Provides a stateful protocol
- Too advanced for its own good
  - Much more complicated then v3
    - Slow adoption
  - Barely being phased in now
    - With hacks that lose some of the features (looks more like v3)