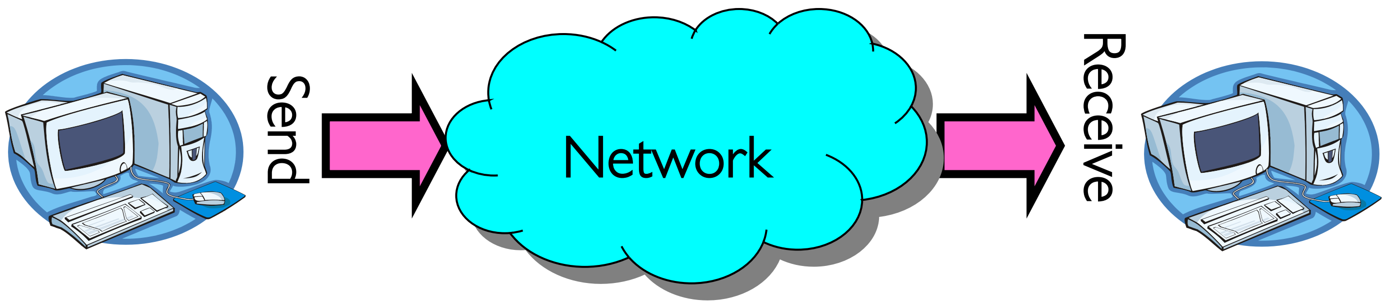
# CS162
# Operating Systems and Systems Programming
# Lecture 24

# RPC,
# Distributed File Systems

# Recall: Distributed Applications Build With Messages

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines




  - One Abstraction: send/receive messages
- Interface:
  - Mailbox (mbox): temporary holding area for messages
  - Send(message,mbox)
    - » Send message to remote mailbox identified by mbox
  - Receive(buffer,mbox)
    - » Wait until mbox has message, copy into buffer, and return

# Question: Data Representation

- An object in memory has a machine-specific binary representation
  - Threads within a single process have the same view of what's in memory
  - Easy to compute offsets into fields, follow pointers, etc.

- In the absence of shared memory, externalizing an object requires us to turn it into a sequential sequence of bytes

  - **Serialization/Marshalling**: Express an object as a sequence of bytes

  - **Deserialization/Unmarshalling**: Reconstructing the original object from its marshalled form at destination

# Simple Data Types

```
uint32_t x;
```

- Suppose I want to write a `x` to a file

- First, open the file: `FILE* f = fopen("foo.txt", "w");`
- Then, I have two choices:
  1. `fprintf(f, "%lu", x);`
  2. `fwrite(&x, sizeof(uint32_t), 1, f);`

- Neither one is "wrong" but sender and receiver should be consistent!

# Machine Representation

- Consider using the machine representation:
  - `fwrite(&x, sizeof(uint32_t), 1, f);`
- How do we know if the recipient represents **`x`** in the same way?
  - For pipes, is this a problem?
  - What about for sockets?

# Endianness

- For a byte-address machine, which end of a machine-recognized object (e.g., int) does its byte-address refer to?
- Big Endian: address is the most-significant bits
- Little Endian: address is the least-significant bits

| Processor | Endianness |
|---|---|
| Motorola 68000 | Big Endian |
| PowerPC (PPC) | Big Endian |
| Sun Sparc | Big Endian |
| IBM S/390 | Big Endian |
| Intel x86 (32 bit) | Little Endian |
| Intel x86_64 (64 bit) | Little Endian |
| Dec VAX | Little Endian |
| Alpha | Bi (Big/Little) Endian |
| ARM | Bi (Big/Little) Endian |
| IA-64 (64 bit) | Bi (Big/Little) Endian |
| MIPS | Bi (Big/Little) Endian |

```
int main(int argc, char *argv[])
{
  int val = 0x12345678;
  int i;
  printf("val = %x\n", val);
  for (i = 0; i < sizeof(val); i++) {
    printf("val[%d] = %x\n", i, ((uint8_t *) &val)[i]);
  }
}
```

```
(base) CullerMac19:code09 culler$ ./endian
val = 12345678
val[0] = 78
val[1] = 56
val[2] = 34
val[3] = 12
```

# What Endian is the Internet?

***NAME***

arpa/inet.h - definitions for internet operations

***SYNOPSIS***

```
#include <arpa/inet.h>
```

***DESCRIPTION***

The **in_port_t** and **in_addr_t** types shall be defined as described in *<netinet/in.h>*.

The **in_addr** structure shall be defined as described in *<netinet/in.h>*.

The INET_ADDRSTRLEN [IP6] ⌦ and INET6_ADDRSTRLEN ⌫ macros shall be defined as described in *<netinet/in.h>*.

The following shall either be declared as functions, defined as macros, or both. If functions are declared, function prototypes

```
uint32_t htonl(uint32_t);
uint16_t htons(uint16_t);
uint32_t ntohl(uint32_t);
uint16_t ntohs(uint16_t);
```

The **uint32_t** and **uint16_t** types shall be defined as described in *<inttypes.h>*.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
in_addr_t    inet_addr(const char *);
char        *inet_ntoa(struct in_addr);
const char  *inet_ntop(int, const void *restrict, char *restrict,
                 socklen_t);
int          inet_pton(int, const char *restrict, void *restrict);
```

Inclusion of the *<arpa/inet.h>* header may also make visible all symbols from *<netinet/in.h>* and *<inttypes.h>*.

- **Big Endian**
  - **Network byte order**
  - **Vs. "host byte order"**

# Dealing with Endianness

- Decide on an "on-wire" endianness
- Convert from native endianness to "on-wire" endianness before sending out data (serialization/marshalling)
  - `uint32_t htonl(uint32_t)` and `uint16_t htons(uint16_t)` convert from native endianness to network endianness (big endian)
- Convert from "on-wire" endianness to native endianness when receiving data (deserialization/unmarshalling)
  - `uint32_t ntohl(uint32_t)` and `uint16_t ntohs(uint16_t)` convert from network endianness to native endianness (big endian)

# What About Richer Objects?

- Consider **`word_count_t`** of Homework 0 and 1 …

```
typedef struct word_count
{
  char *word;
  int count;
  struct word_count *next;
}
word_count_t;
```

- Each element contains:
  - An **`int`**
  - A *pointer* to a string (of some length)
  - A *pointer* to the next element

- **`fprintf_words`** writes these as a sequence of lines (character strings with **`\n`**) to a file stream

- What if you wanted to write the whole list as a binary object (and read it back as one)?
  - How do you represent the string?
  - Does it make any sense to write the pointer?

# Data Serialization Formats

- Google Protobuffers, JSON and XML are commonly used in web applications
- Lots of ad-hoc formats

```
{ “ faculty”:
      [
              {id: 1,
               “name”: “Anthony”,
               “lastname”: “Joseph”
               },
              {id: 2,
               “name”: “Natacha”,
               “lastname”: “Crooks”
              }
      ]
}
```

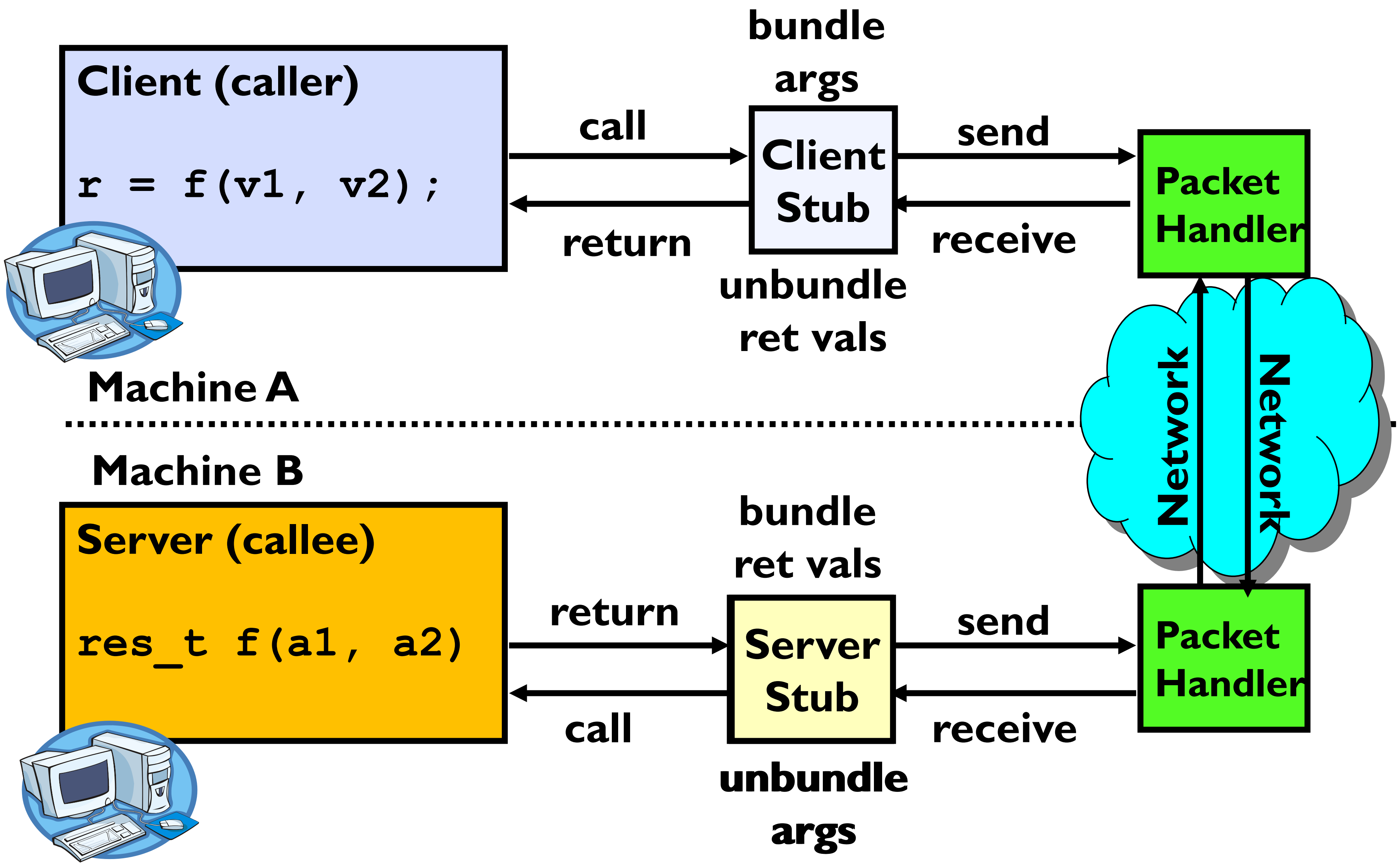
# Data Serialization Formats

| Name | Creator-maintainer | Based on | Standardized? | Specification | Binary? | Human-readable? | Supports references?[e] | Schema-IDL? | Standard APIs | Supports Zero-copy operations |
|---|---|---|---|---|---|---|---|---|---|---|
| Apache Avro | Apache Software Foundation | N/A | No | Apache Avro™ 1.8.1 Specification | Yes | No | N/A | Yes (built-in) | N/A | N/A |
| Apache Parquet | Apache Software Foundation | N/A | No | Apache Parquet[1] | Yes | No | No | N/A | Java, Python | No |
| ASN.1 | ISO, IEC, ITU-T | N/A | Yes | ISO/IEC 8824; X.680 series of ITU-T Recommendations | Yes (BER, DER, PER, OER, or custom via ECN) | Yes (XER, JER, GSER, or custom via ECN) | Partial[f] | Yes (built-in) | N/A | Yes (OER) |
| Bencode | Bram Cohen (creator) BitTorrent, Inc. (maintainer) | N/A | *De facto* standard via BitTorrent Enhancement Proposal (BEP) | Part of BitTorrent protocol specification | Partially (numbers and delimiters are ASCII) | No | No | No | No | N/A |
| Binn | Bernardo Ramos | N/A | No | Binn Specification | Yes | No | No | No | No | Yes |
| BSON | MongoDB | JSON | No | BSON Specification | Yes | No | No | No | No | N/A |
| CBOR | Carsten Bormann, P. Hoffman | JSON (loosely) | Yes | RFC 7049 | Yes | No | Yes through tagging | Yes (CDDL) | No | Yes |
| Comma-separated values (CSV) | RFC author: Yakov Shafranovich | N/A | Partial (myriad informal variants used) | RFC 4180 (among others) | No | Yes | No | No | No | No |
| Common Data Representation (CDR) | Object Management Group | N/A | Yes | General Inter-ORB Protocol | Yes | No | Yes | Yes | ADA, C, C++, Java, Cobol, Lisp, Python, Ruby, Smalltalk | N/A |
| D-Bus Message Protocol | freedesktop.org | N/A | Yes | D-Bus Specification | Yes | No | No | Partial (Signature strings) | Yes (see D-Bus) | N/A |
| Efficient XML Interchange (EXI) | W3C | XML, Efficient XML | Yes | Efficient XML Interchange (EXI) Format 1.0 | Yes | Yes (XML) | Yes (XPointer, XPath) | Yes (XML Schema) | Yes (DOM, SAX, StAX, XQuery, XPath) | N/A |
| FlatBuffers | Google | N/A | No | flatbuffers github page Specification | Yes | Yes (Apache Arrow) | Partial (internal to the buffer) | Yes [2] | C++, Java, C#, Go, Python, Rust, JavaScript, PHP, C, Dart, Lua, TypeScript | Yes |
| Fast Infoset | ISO, IEC, ITU-T | XML | Yes | ITU-T X.891 and ISO/IEC 24824-1:2007 | Yes | No | Yes (XPointer, XPath) | Yes (XML schema) | Yes (DOM, SAX, XQuery, XPath) | N/A |
| FHIR | Health_Level_7 | REST basics | Yes | Fast Healthcare Interoperability Resources | Yes | Yes | Yes | Yes | Hapi for FHIR[1] JSON, XML, Turtle | No |
| Ion | Amazon | JSON | No | The Amazon Ion Specification | Yes | Yes | No | No | No | N/A |
| Java serialization | Oracle Corporation | N/A | Yes | Java Object Serialization | Yes | No | Yes | No | Yes | N/A |
| JSON | Douglas Crockford | JavaScript syntax | Yes | STD 90/RFC 8259 (ancillary: RFC 6901, RFC 6902), ECMA-404, ISO/IEC 21778:2017 | No, but see BSON, Smile, UBJSON | Yes | Yes (JSON Pointer (RFC 6901); alternately: JSONPath, JPath, JSPON, json:select()), JSON-LD | Partial (JSON Schema Proposal, ASN.1 with JER, Kwalify, Rx, Itemscript Schema), JSON-LD | Partial (Clarinet, JSONQuery, JSONPath), JSON-LD | No |
| MessagePack | Sadayuki Furuhashi | JSON (loosely) | No | MessagePack format specification | Yes | No | No | No | No | Yes |
| Netstrings | Dan Bernstein | N/A | No | netstrings.txt | Yes | Yes | No | No | No | Yes |
| OGDL | Rolf Veen | ? | No | Specification | Yes (Binary Specification) | Yes | Yes (Path Specification) | Yes (Schema WD) | | N/A |
| OPC-UA Binary | OPC Foundation | N/A | No | opcfoundation.org | Yes | No | Yes | No | No | N/A |
| OpenDDL | Eric Lengyel | C, PHP | No | OpenDDL.org | No | Yes | Yes | No | Yes (OpenDDL Library) | N/A |
| Pickle (Python) | Guido van Rossum | Python | *De facto* standard via Python Enhancement Proposals (PEPs) | [3] PEP 3154 -- Pickle protocol version 4 | Yes | No | No | No | Yes ([4]) | No |
| Property list | NeXT (creator) Apple (maintainer) | ? | Partial | Public DTD for XML format | Yes[a] | Yes[b] | No | ? | Cocoa, CoreFoundation, OpenStep, GnuStep | No |
| Protocol Buffers (protobuf) | Google | N/A | No | Developer Guide: Encoding | Yes | Partial[d] | No | Yes (built-in) | C++, C#, Java, Python, Javascript, Go | No |
| | John McCarthy | | | | | Yes | | | | |

# Remote Procedure Call (RPC)

- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
  - And must deal with machine representation by hand
- Another option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Idea: Make communication look like an ordinary function call
  - Automate all of the complexity of translating between representations
  - Client calls:
    ```
    remoteFileSystem→Read("rutabaga");
    ```
  - Translated automatically into call on server:
    ```
    fileSys→Read("rutabaga");
    ```
- Example: Java RMI

# RPC Information Flow



# RPC Implementation

- Request-response message passing (under covers!)

- "Stub" provides glue on client/server
  - Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
  - Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.

- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

# RPC Details (1/3)

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)

- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    - » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - » Code for server to unpack message, call procedure, pack results, send them off

# RPC Details (2/3)

- Cross-platform issues:
  - What if client/server machines are different architectures/ languages?
    - » Convert everything to/from some canonical form
    - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions)
- How does client know which mbox (destination queue) to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    - » This is another word for "naming" at network level
    - » Static: fixed at compile time
    - » Dynamic: performed at runtime

# RPC Details (3/3)

- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynamic translation of service → mbox
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one

- What if there are multiple servers?
  - Could give flexibility at binding time
    - » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - » Choose unloaded server for each new request
    - » Only works if no state carried from one call to next

- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

# Problems with RPC: Non-Atomic Failures

- Different failure modes in dist. system than on a single machine

- Consider many different types of failures
  - User-level bug causes address space to crash
  - Machine failure, kernel bug causes all processes on same machine to fail
  - Some machine is compromised by malicious party
- Before RPC: whole system would crash/die
- After RPC: One machine crashes/compromised while others keep working

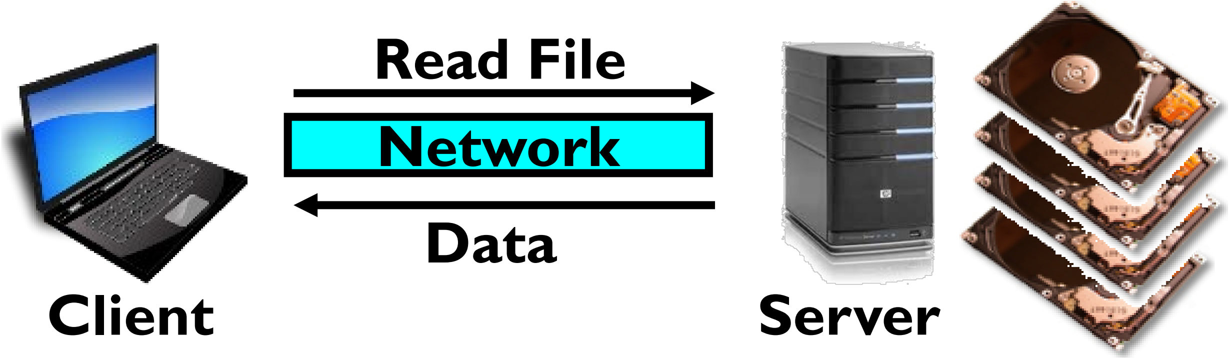
- Can easily result in inconsistent view of the world
  - Did my cached data get written back or not?
  - Did server do what I requested or not?

- Answer? Distributed transactions/2PC

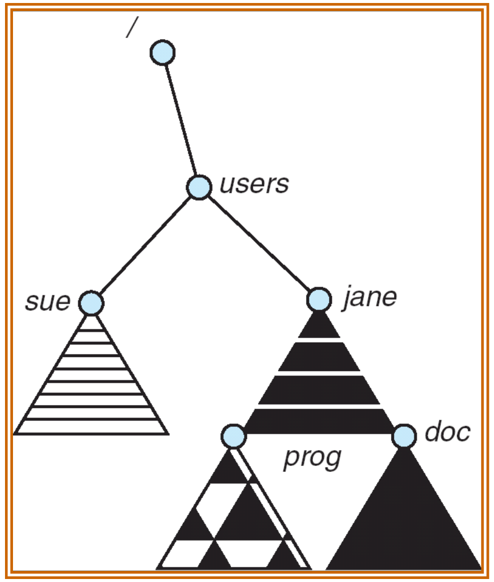
# Problems with RPC: Performance

- RPC is *not* performance transparent:
  - Cost of Procedure call « same-machine RPC « network RPC
  - Overheads: Marshalling, Stubs, Kernel-Crossing, Communication
- Programmers must be aware that RPC is not free
  - Caching can help, but may make failure handling complex

# Distributed File Systems

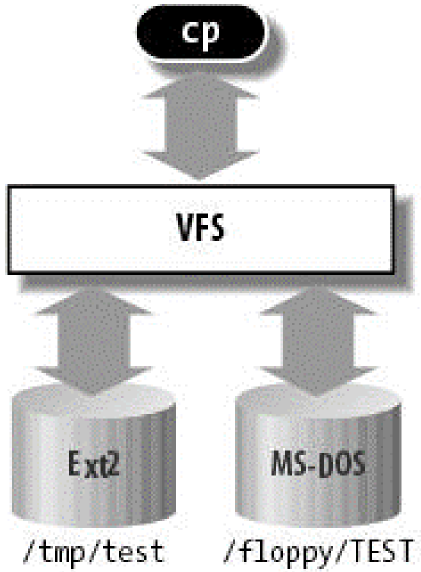



- Transparent access to files stored on a remote disk
- *Mount* remote files into your local file system
  – Directory in local file system refers to remote files
  – e.g., `/users/jane/prog/foo.c` on laptop actually refers to `/prog/foo.c` on `crooks.cs.berkeley.edu`




- *Naming* Choices:
  – [Hostname,localname]: Filename includes server
  – A global name space: Filename unique in "world"

# Virtual Filesystem Switch

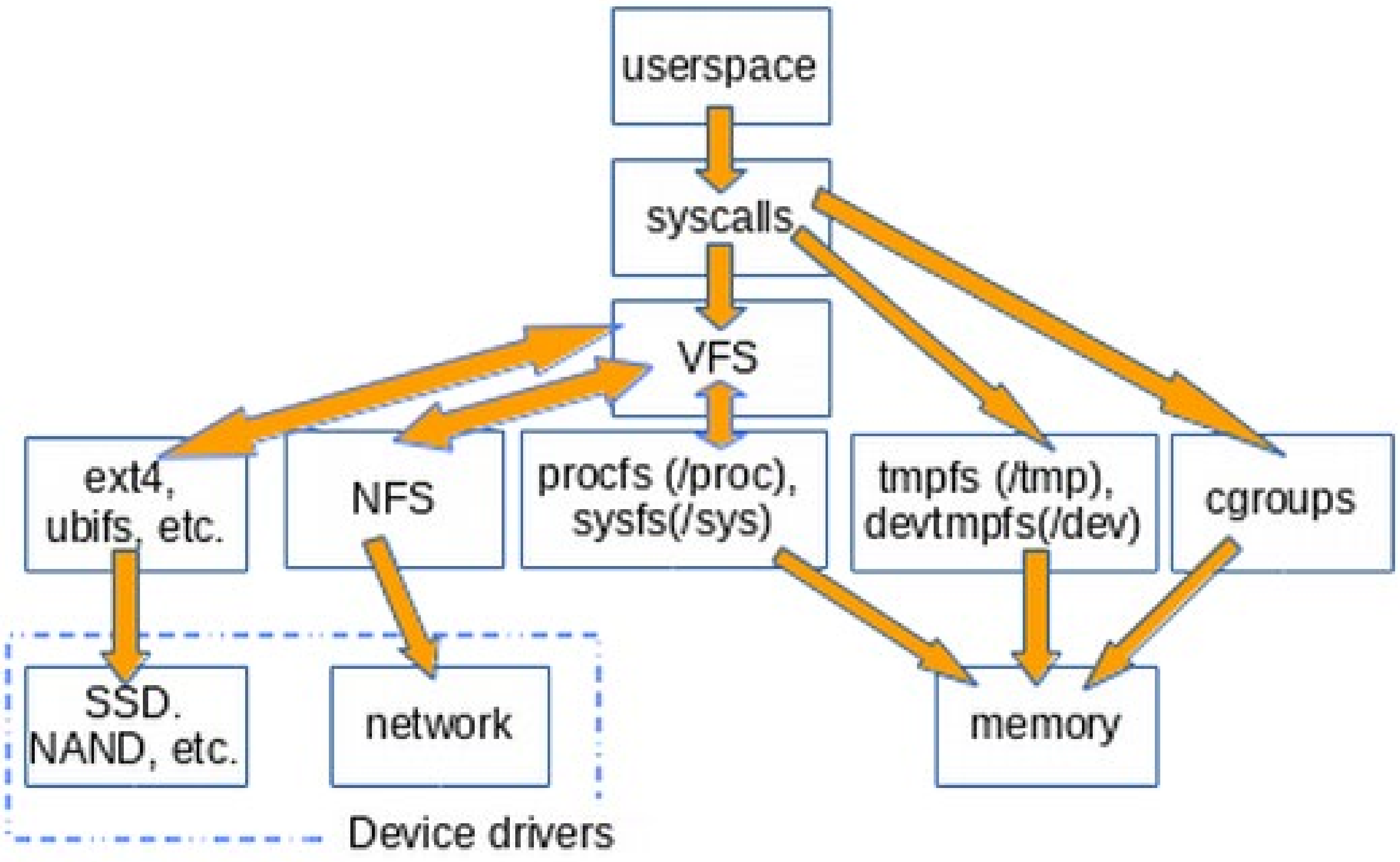



```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

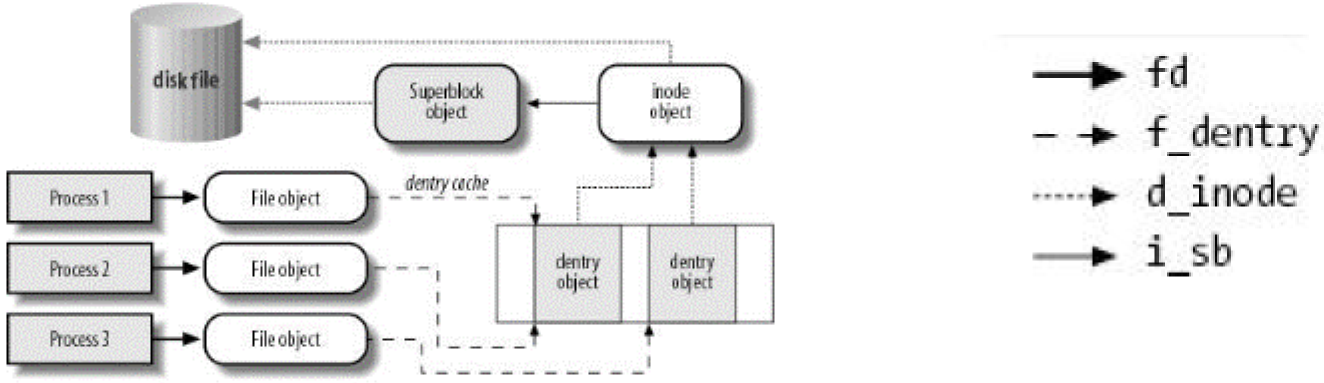
- VFS: Virtual abstraction of file system
  – Provides virtual superblocks, inodes, files, etc
  – Compatible with a variety of local and remote file systems

- VFS allows the same system call interface (the API) to be used for different types of file systems
  – The API is to the VFS interface, rather than any specific type of file system

# Example Linux mouting tree



# VFS Common File Model in Linux

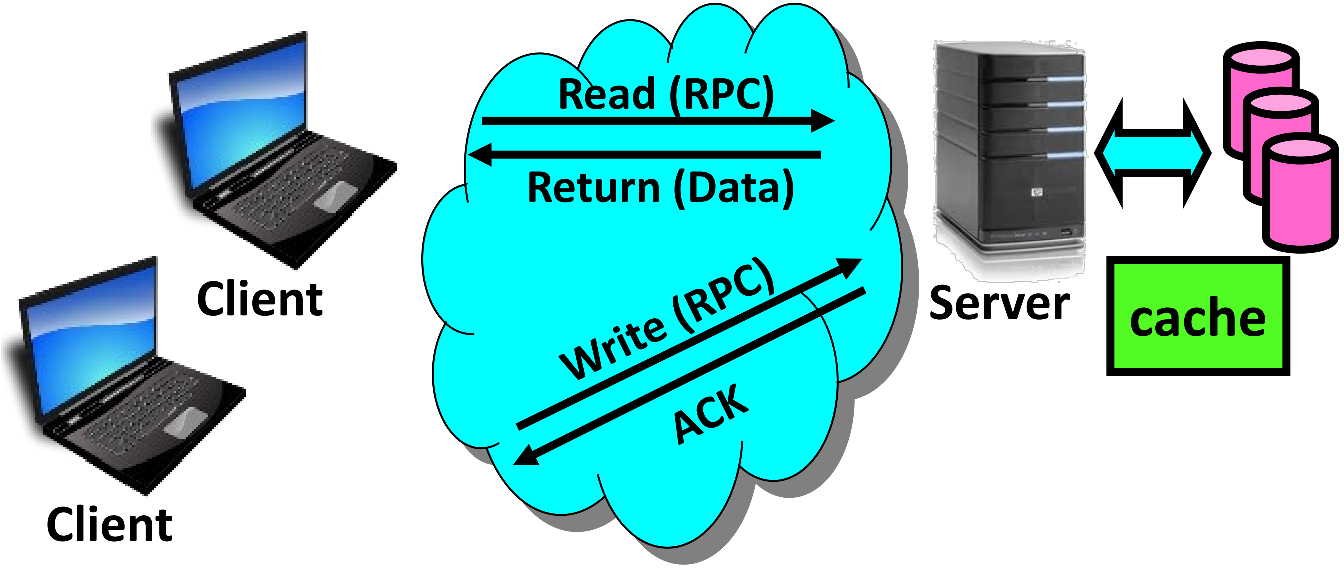



- Four primary object types for VFS:
  - superblock object: represents a specific mounted filesystem
  - inode object: represents a specific file
  - dentry object: represents a directory entry
  - file object: represents open file associated with process

- There is no specific directory object (VFS treats directories as files)

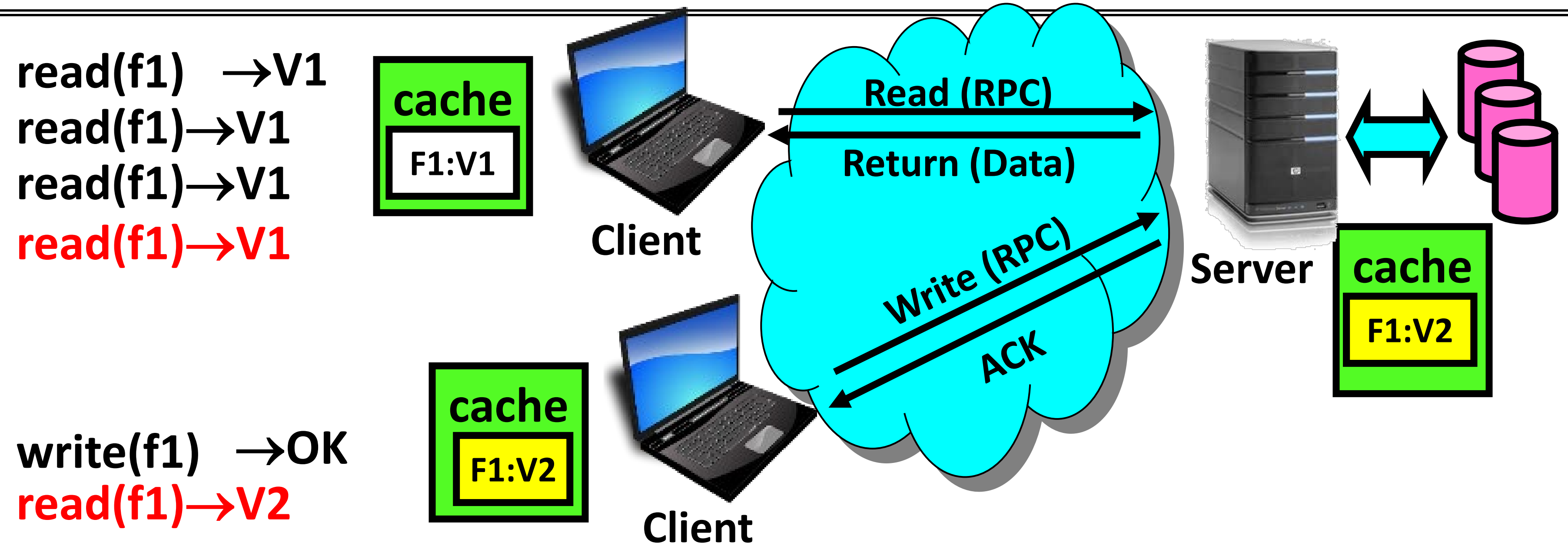
- May need to fit the model by faking it

# Simple Distributed File System




- Remote Disk: Reads and writes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
- Advantage: Server provides consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic/not well pipelined
  - Server can be a bottleneck

# Use of caching to reduce network load




- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - » Client caches have data not committed at server
  - Cache consistency!
    - » Client caches not consistent with server/each other

# Dealing with Failures

- What if server crashes? Can client wait until it comes back and just continue making requests?
  - Changes in server's cache but not in disk are lost
- What if there is shared state across RPC's?
  - Client opens file, then does a seek
  - Server crashes
  - What if client wants to do another read?
- Similar problem: What if client removes a file but server crashes before acknowledgement?

# Stateless Protocol

- Stateless Protocol: A protocol in which all information required to service a request is included with the request
- Even better: Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)
- Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)
- Recall HTTP: Also a stateless protocol
  - Include cookies with request to simulate a session

# Case Study: Network File System (NFS)

- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol

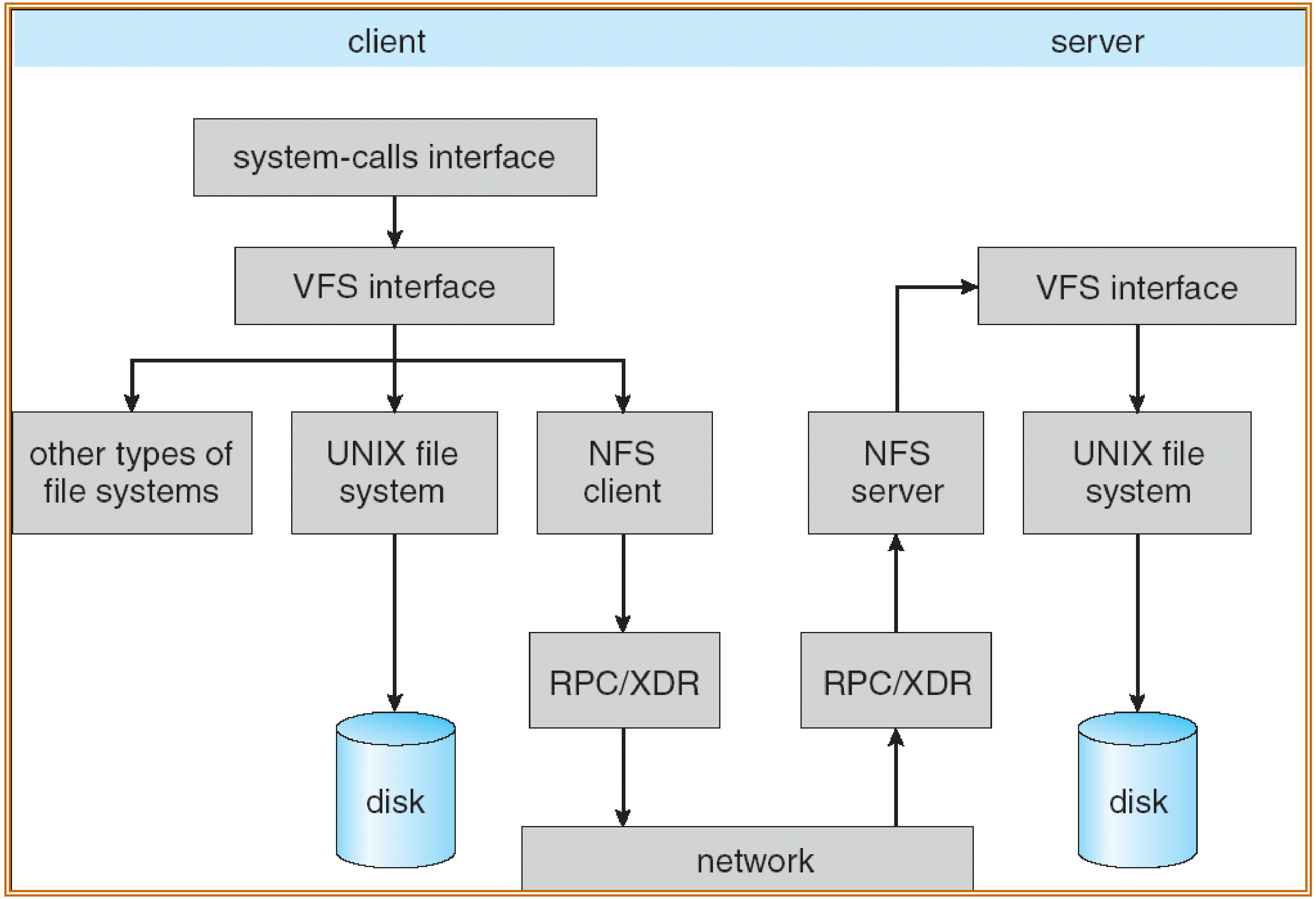
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files

- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes! (more on this later)

# NFS Continued

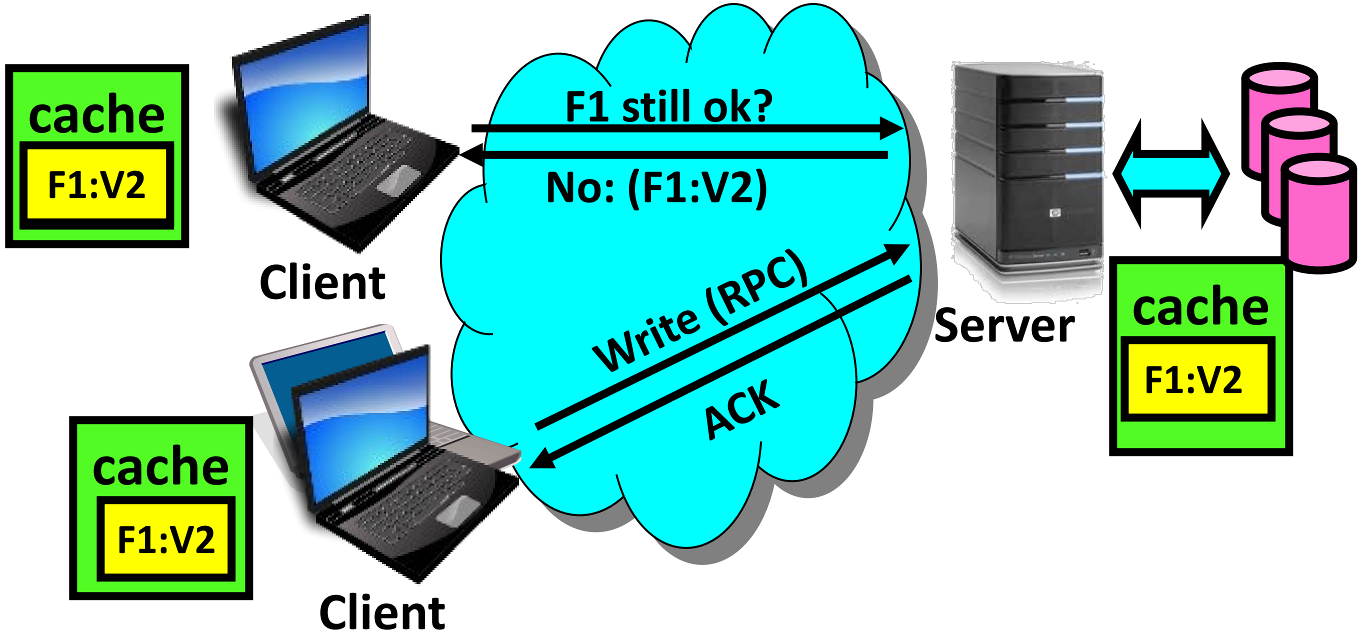
- NFS servers are stateless; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as **`ReadAt(inumber,position)`**, not **`Read(openfile)`**
  - No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing them exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no other side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error
- Failure Model: Transparent to client system
  - Is this a good idea? What if you are in the middle of reading a file and server crashes?
  - Options (NFS Provides both):
    - » Hang until server comes back up (next week?)
    - » Return an error. (Of course, most applications don't know they are talking over network)

# NFS Architecture



# NFS Cache consistency

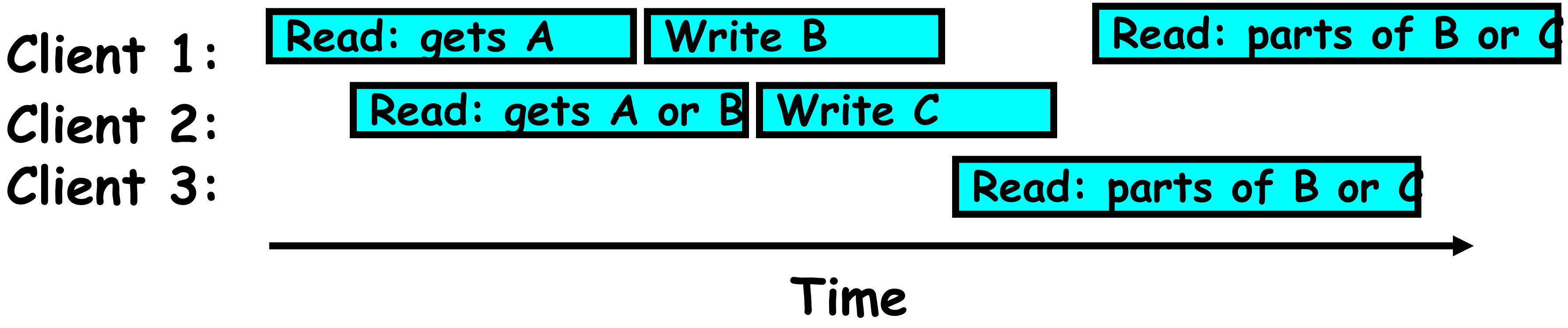
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.




  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

# Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, and before it's done, another CPU reads file?
- Example: Start with file contents = "A"




- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - » If read finishes before write starts, get old copy
    - » If read starts after write finishes, get new copy
    - » Otherwise, get either new or old copy
  - For NFS:
    - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

# NFS Pros and Cons

- NFS Pros:
  - Simple, Highly portable
- NFS Cons:
  - Sometimes inconsistent!
  - Doesn't scale to large # clients
    - » Must keep checking to see if caches out of date
    - » Server becomes bottleneck due to polling traffic

# Summary

- Message passing and the challenges of serialization/deserialization
- Remote Procedure Calls: abstraction of local computation on remote machines
- Distributed File Systems using VFS
  - NFS: weak consistency but efficient