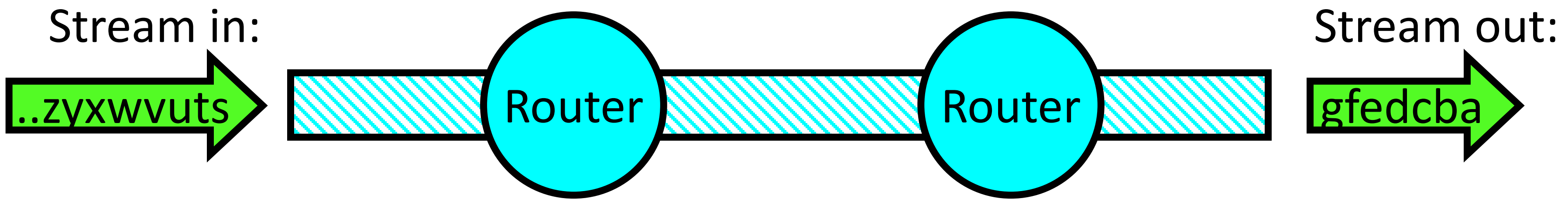
CSC 112: Computer Operating Systems
Lecture 25

RPC, NFS and AFS

Department of Computer Science,
Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming

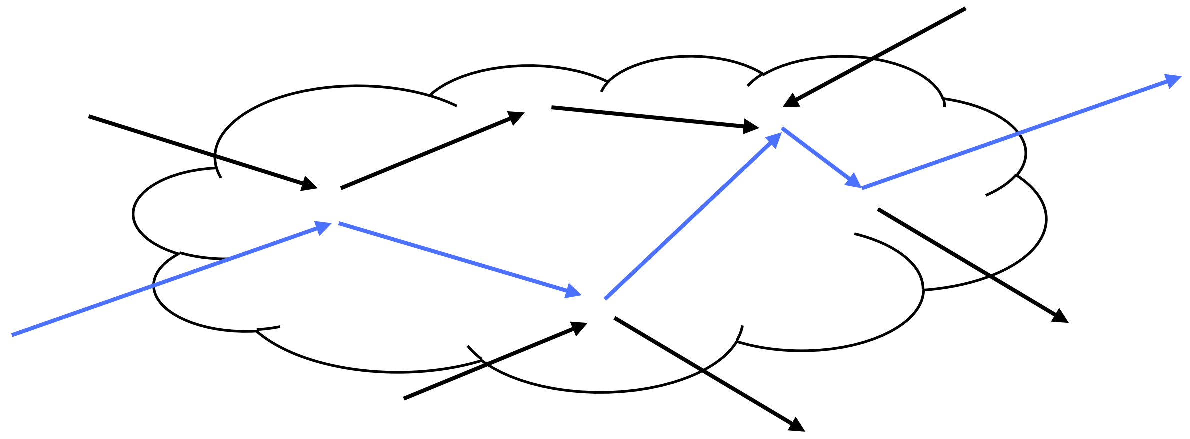
Recall: Transmission Control Protocol (TCP)



- Transmission Control Protocol (TCP)
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- TCP Details
  - Fragments byte stream into packets, hands packets to IP
    - » IP may also fragment by itself
  - Uses window-based acknowledgement protocol (to minimize state at sender and receiver)
    - » "Window" reflects storage at receiver – sender shouldn't overrun receiver's buffer space
    - » Also, window should reflect speed/capacity of network – sender shouldn't overload network
  - Automatically retransmits lost packets
  - Adjusts rate of transmission to avoid congestion
    - » A "good citizen"

Congestion

- Too much data trying to flow through some part of the network

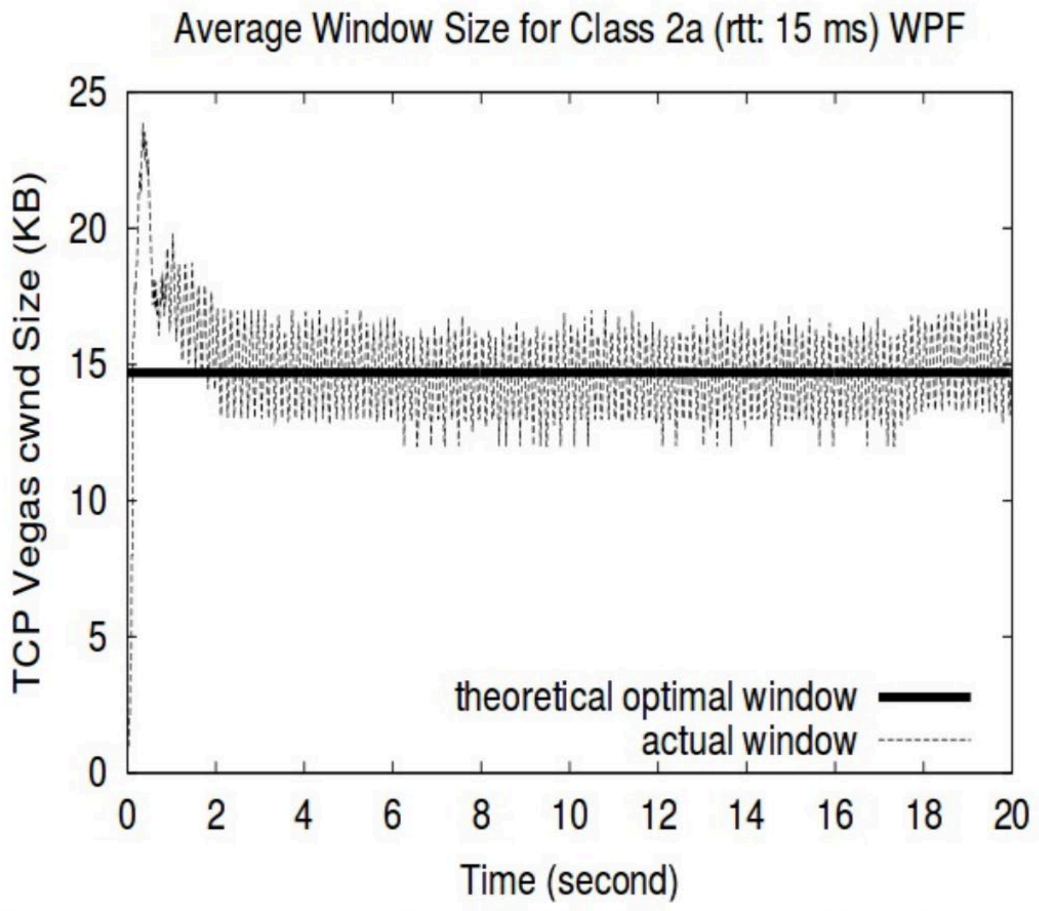


- IP's solution: Drop packets
- What happens to TCP connection?
  - Lots of retransmission – wasted work and wasted bandwidth (when bandwidth is scarce)

Congestion Avoidance

- Congestion
  - How long should timeout be for re-sending messages?
    - » Too long $\rightarrow$ wastes time if message lost
    - » Too short $\rightarrow$ retransmit even though ACK will arrive shortly
  - Stability problem: more congestion $\Rightarrow$ ACK is delayed $\Rightarrow$ unnecessary timeout $\Rightarrow$ more traffic $\Rightarrow$ more congestion
    - » Closely related to window size at sender: too big means putting too much data into network
- How does the sender's window size get chosen?
  - Must be less than receiver's advertised buffer size
  - Try to match the rate of sending packets with the rate that the slowest link can accommodate
  - Sender uses an adaptive algorithm to decide size of N
    - » Goal: fill network between sender and receiver
    - » Basic technique: slowly increase size of window until acknowledgements start being delayed/lost
- TCP solution: "slow start" (start sending slowly)
  - If no timeout, slowly increase window size (throughput) by 1 for each ACK received
  - Timeout $\Rightarrow$ congestion, so cut window size in half
  - "*Additive Increase, Multiplicative Decrease*"

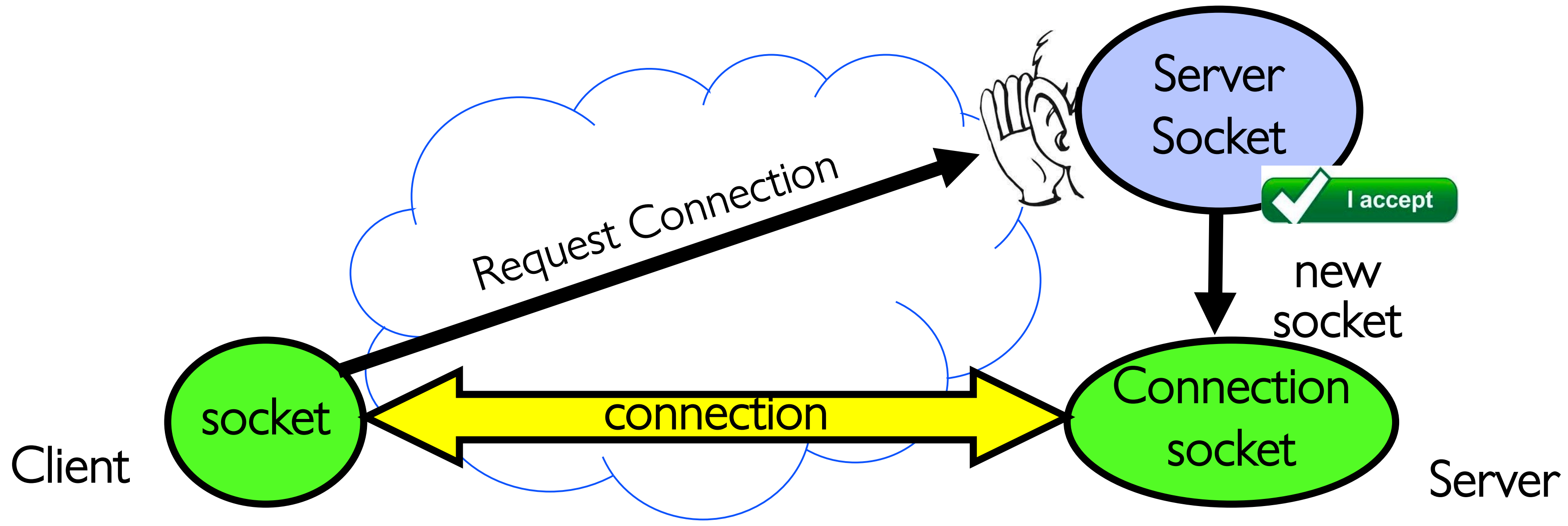
Congestion Management



From Low, Peterson, and Wang, "Understanding vegas: Duality Model", J. ACM, March 2002.

- TCP artificially restricts the window size if it sees packet loss
- Careful control loop to make sure:
  1. We don't send too fast and overwhelm the network
  2. We utilize most of the bandwidth the network has available
  - – In general, these are conflicting goals!

Recall: Connection Setup over TCP/IP



- 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)

- Often, Client Port "randomly" assigned
  – Done by OS during client socket setup
- Server Port often "well known"
  – 80 (web), 443 (secure web), 25 (sendmail), etc
  – Well-known ports from 0—1023

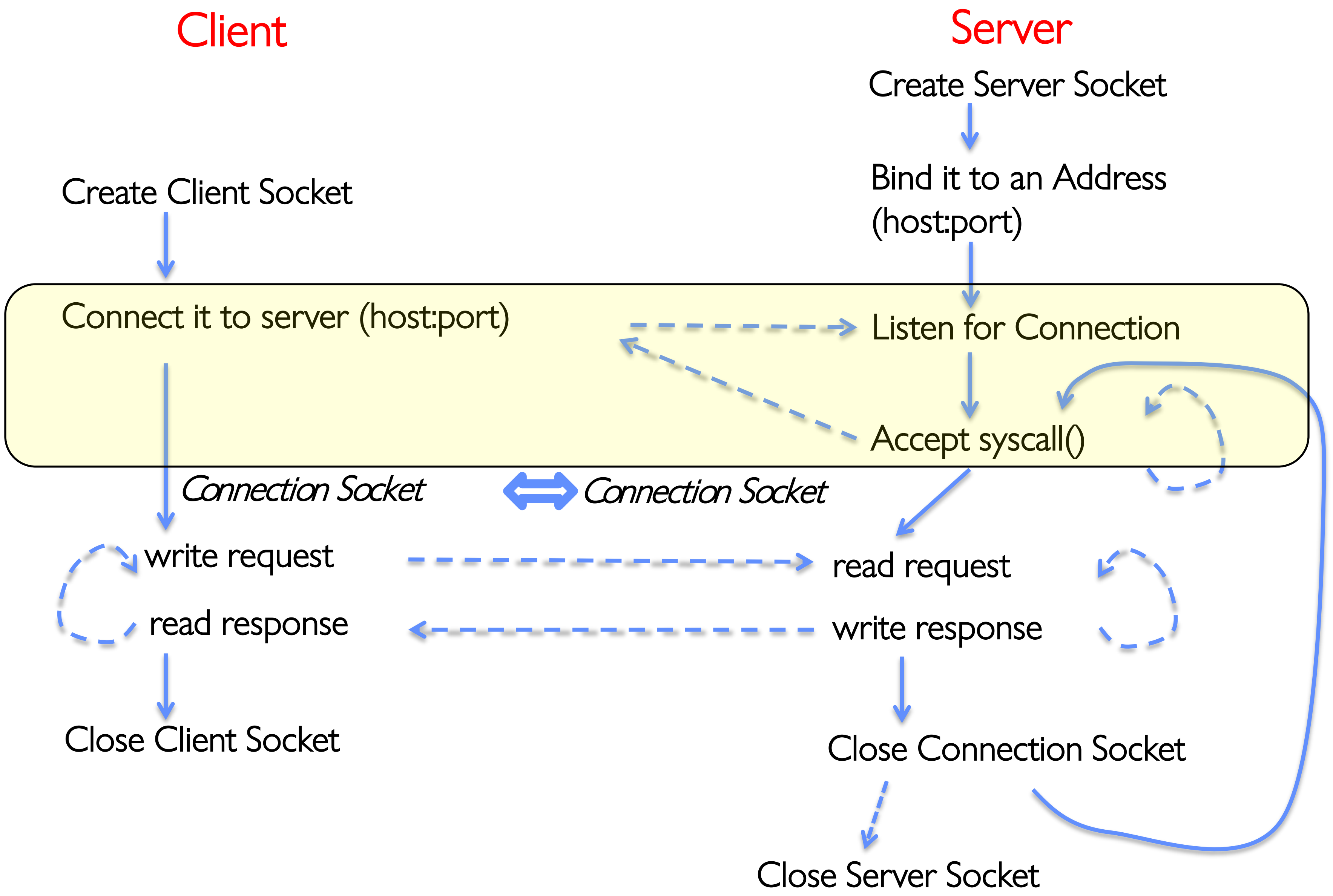
Establishing TCP Service

1. Open connection: 3-way handshaking

2. Reliable byte stream transfer from (IPa, TCP_Port1) to (IPb, TCP_Port2)
   – Indication if connection fails: Reset

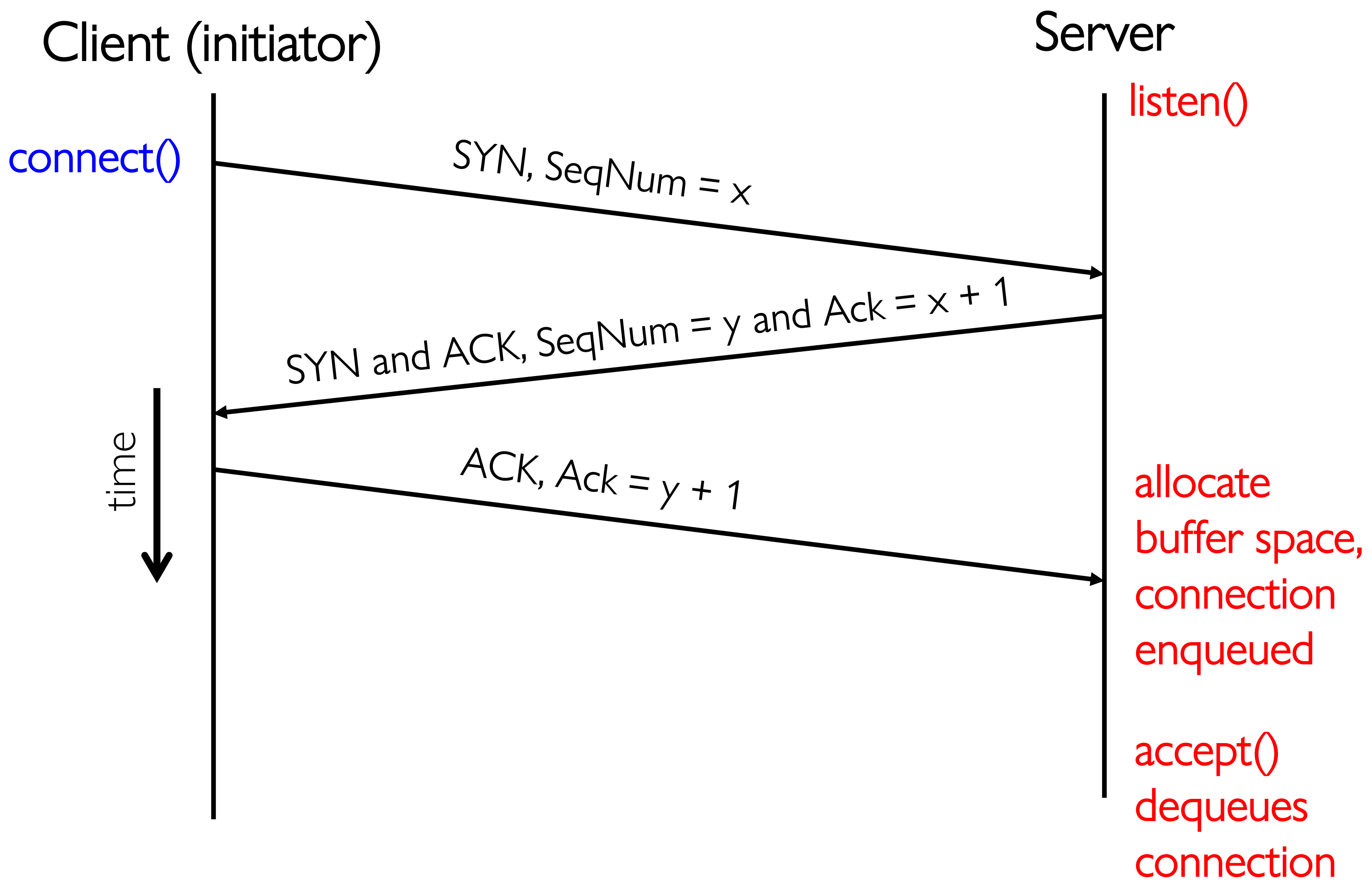
3. Close (tear-down) connection

Sockets in concept

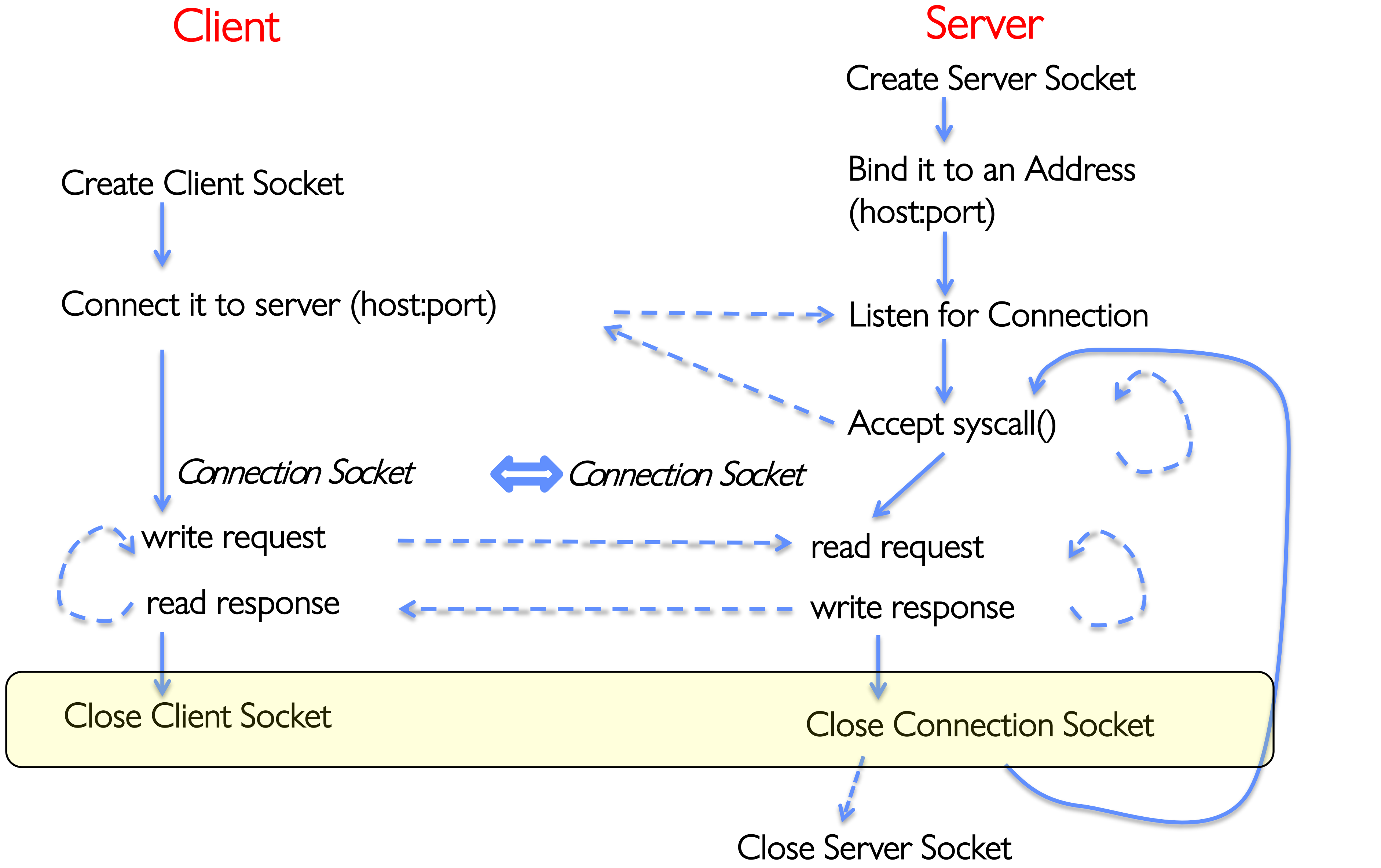


Open Connection: 3-Way Handshake

- Server calls listen() to wait for a new connection
- Client calls connect() providing server's IP address and port number
- Each side sends SYN packet proposing an initial sequence number (one for each sender) and ACKs the other


Client (initiator)
Server
listen()
connect()
SYN, SeqNum = x
SYN and ACK, SeqNum = y and Ack = x + 1
time
ACK, Ack = y + 1
allocate buffer space, connection enqueued
accept() dequeues connection

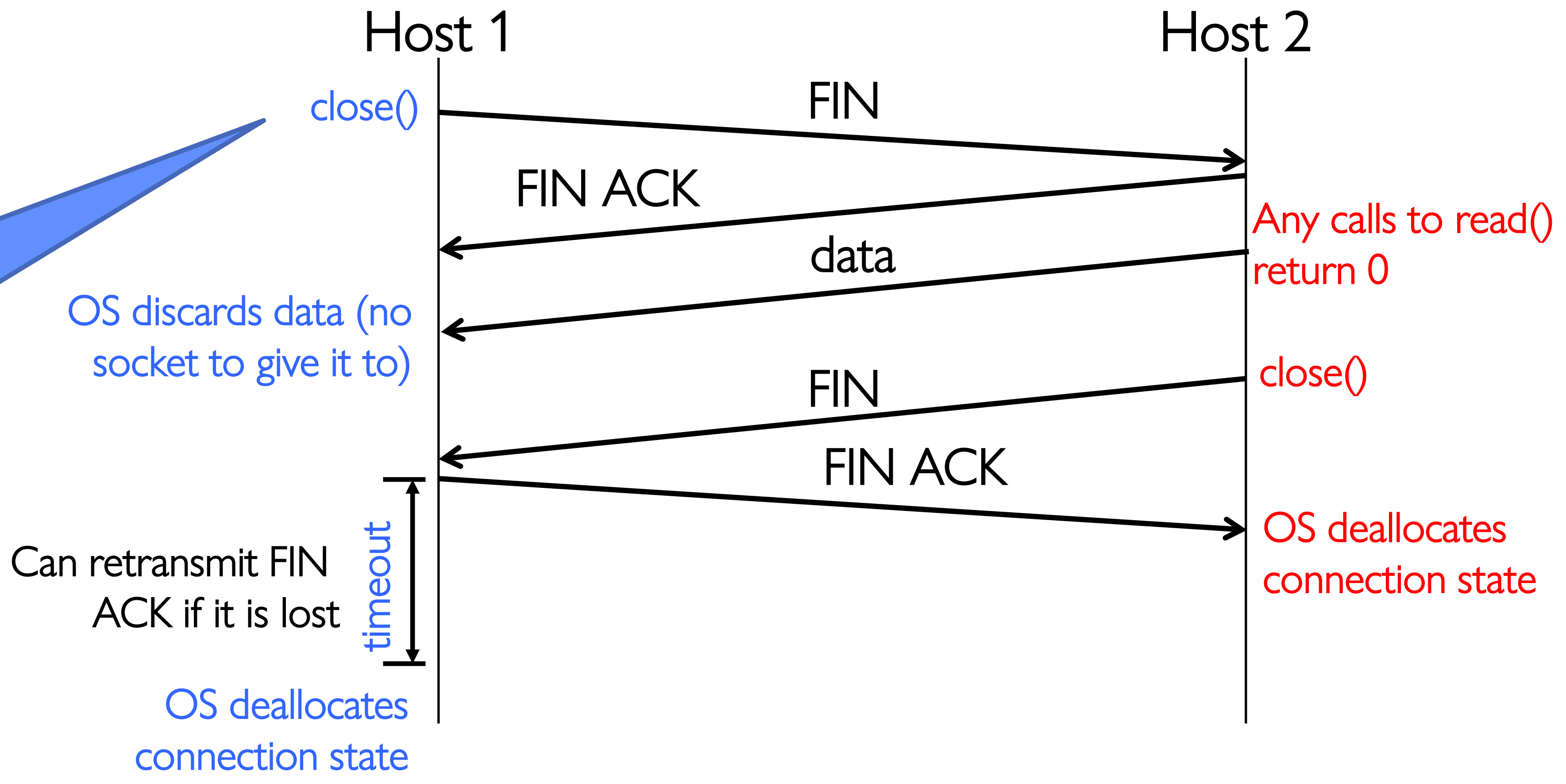
Sockets in concept



Close Connection: 4-Way Teardown

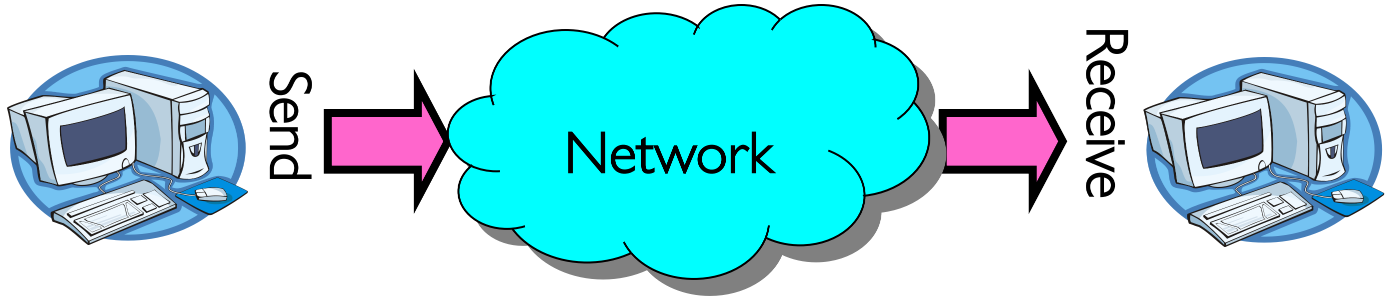
- Connection is not closed until both sides agree

- If multiple FDs on Host 1 refer to this connection, *all* of them must be closed
- Same for close() call on Host 2


Host 1
Host 2
close()
FIN
FIN ACK
Any calls to read() return 0
data
OS discards data (no socket to give it to)
close()
FIN
FIN ACK
OS deallocates connection state
Can retransmit FIN ACK if it is lost
timeout
OS deallocates connection state

Recall: Distributed Applications Build With Messages

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set



  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (mbox): temporary holding area for messages
    - » Includes both destination location and queue
  - Send(message,mbox)
    - » Send message to remote mailbox identified by mbox
  - Receive(buffer,mbox)
    - » Wait until mbox has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them

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);`
     - » Or equivalently, `write(fd, &x, sizeof(uint32_t));` (perhaps with a loop to be safe)

- 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]);
  }
}
```

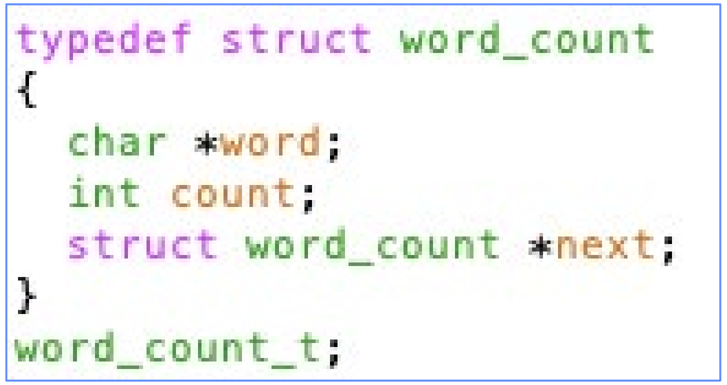
```
(base) CullerMac19:code09 culler$ ./endian
val = 12345678
val[0] = 78
val[1] = 56
val[2] = 34
val[3] = 12
```

Network byte order Vs. “host byte order”

- 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?

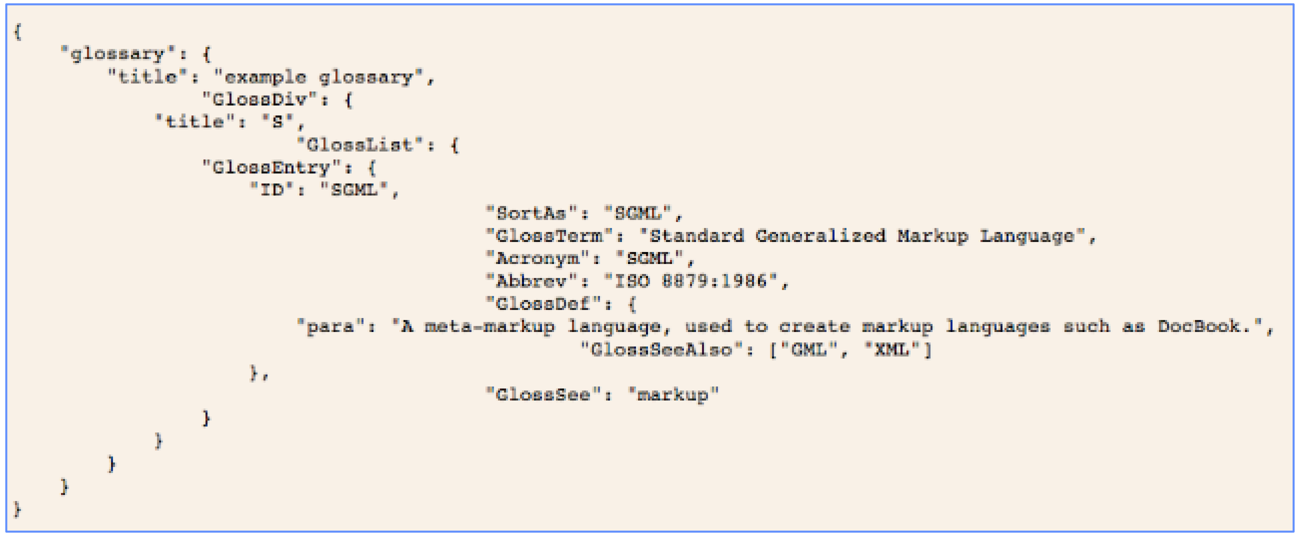


```
typedef struct word_count
{
  char *word;
  int count;
  struct word_count *next;
}
word_count_t;
```

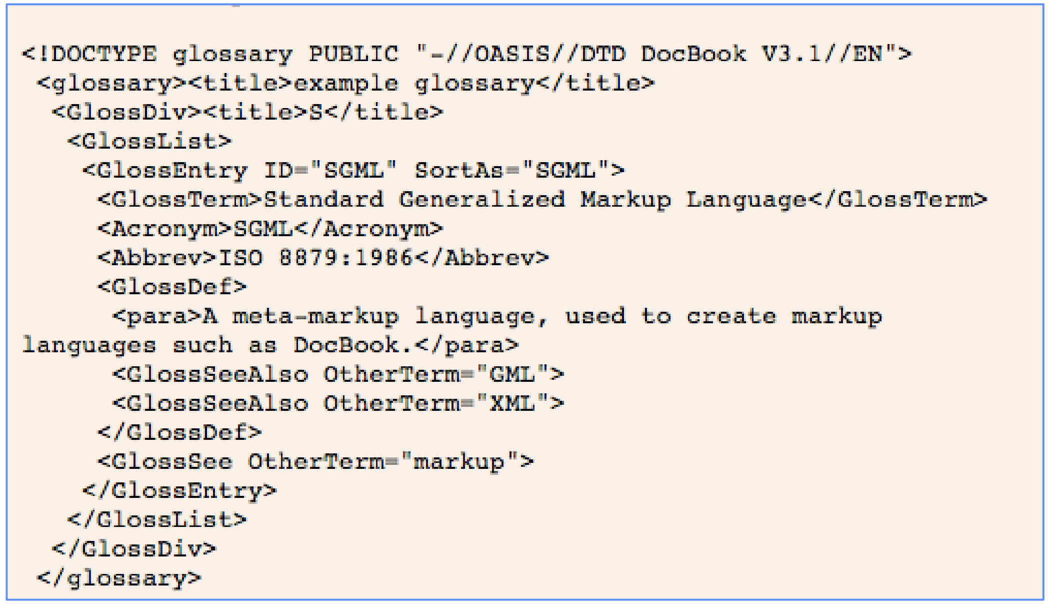
- Consider **`word_count_t`** of Homework 0 and 1 …
- Each element contains:
  - An **`int`**
  - A *pointer* to a string (of some length)
  - A *pointer* to the next element
- **`fprintf_words`** writes these as a sequence of lines (character strings with **`\n`**) to a file stream
- What if you wanted 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

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



```
{
    "glossary": {
        "title": "example glossary",
                "GlossDiv": {
            "title": "S",
                    "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                                        "SortAs": "SGML",
                                        "GlossTerm": "Standard Generalized Markup Language",
                                        "Acronym": "SGML",
                                        "Abbrev": "ISO 8879:1986",
                                        "GlossDef": {
                        "para": "A meta-markup language, used to create markup languages such as DocBook.",
                                                "GlossSeeAlso": ["GML", "XML"]
                    },
                                        "GlossSee": "markup"
                }
            }
        }
    }
}
```



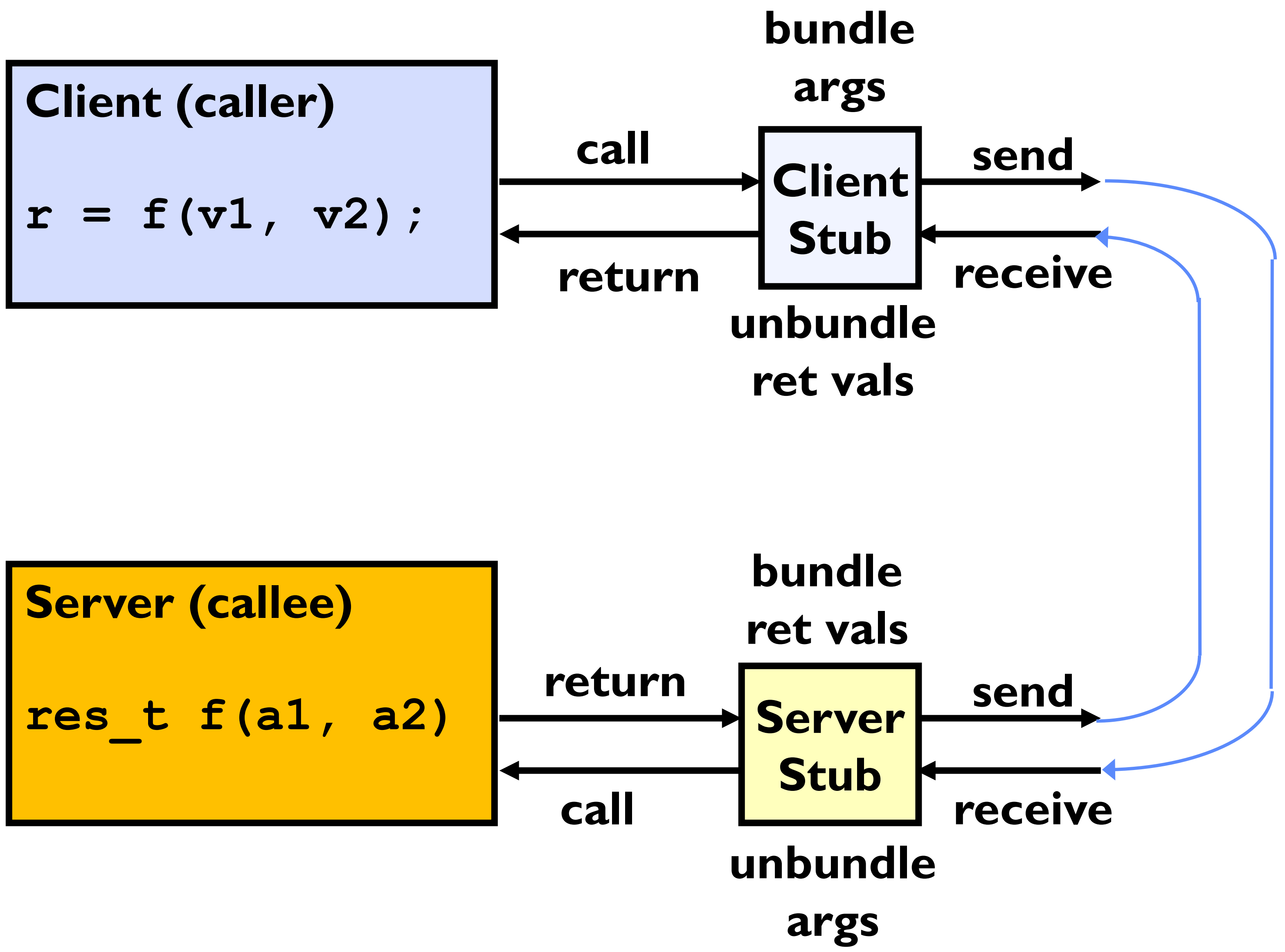
```
<!DOCTYPE glossary PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
 <glossary><title>example glossary</title>
  <GlossDiv><title>S</title>
   <GlossList>
    <GlossEntry ID="SGML" SortAs="SGML">
     <GlossTerm>Standard Generalized Markup Language</GlossTerm>
     <Acronym>SGML</Acronym>
     <Abbrev>ISO 8879:1986</Abbrev>
     <GlossDef>
      <para>A meta-markup language, used to create markup
languages such as DocBook.</para>
      <GlossSeeAlso OtherTerm="GML">
      <GlossSeeAlso OtherTerm="XML">
     </GlossDef>
     <GlossSee OtherTerm="markup">
    </GlossEntry>
   </GlossList>
  </GlossDiv>
 </glossary>
```

Remote Procedure Call (RPC)

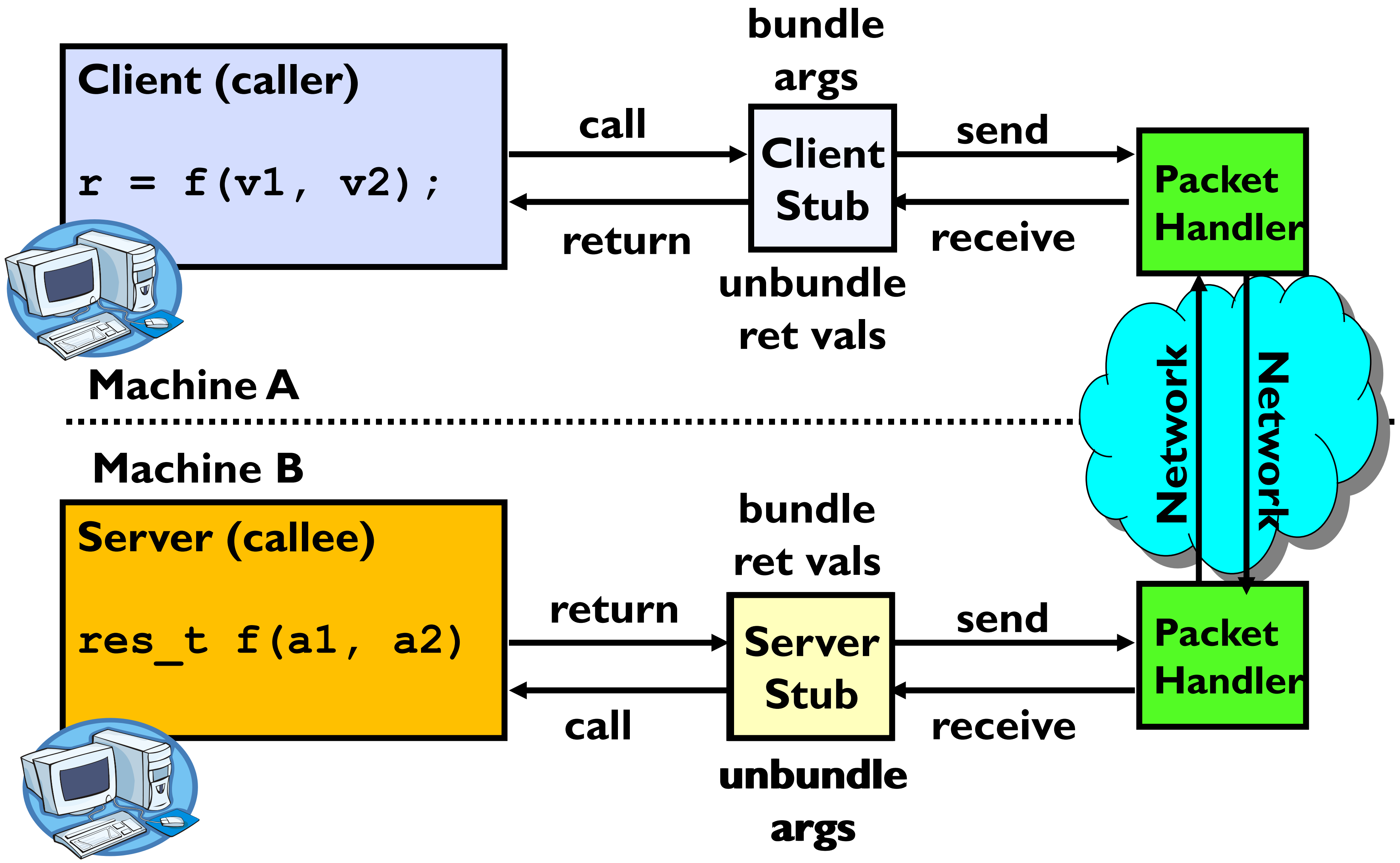
- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
  - And must deal with machine representation by hand

- 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");
    ```

RPC Concept


bundle
args
Client (caller)
r = f(v1, v2);
call
return
Client
Stub
send
receive
unbundle
ret vals
bundle
ret vals
Server (callee)
res_t f(a1, a2)
return
call
Server
Stub
send
receive
unbundle
args

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/Byzantine Commit

Problems with RPC: Performance

- RPC is *not* performance transparent:
  - Cost of Procedure call « same-machine RPC « network RPC
  - Overheads: Marshalling, Stubs, Kernel-Crossing, Communication

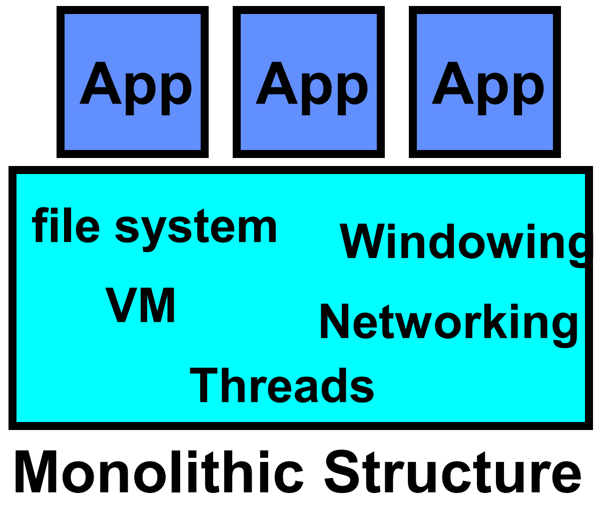
- Programmers must be aware that RPC is not free
  - Caching can help, but may make failure handling complex

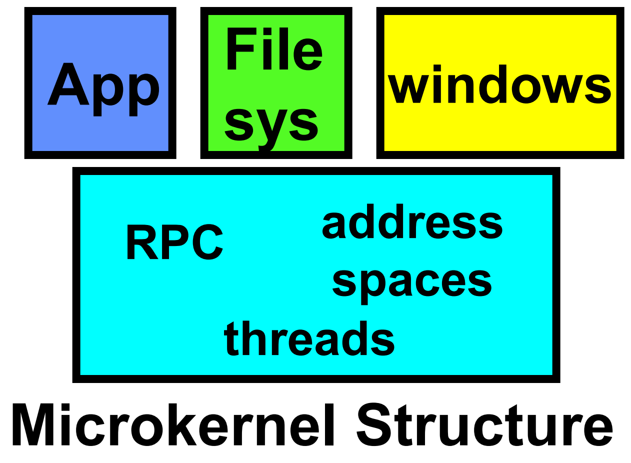
Cross-Domain Communication/Location Transparency

- How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc…
  - File System
  - Pipes (1-way communication)
  - “Remote” procedure call (2-way communication)
- RPC’s can be used to communicate between address spaces on different machines or the same machine
  - Services can be run wherever it’s most appropriate
  - Access to local and remote services looks the same
- Examples of RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)

Microkernel operating systems

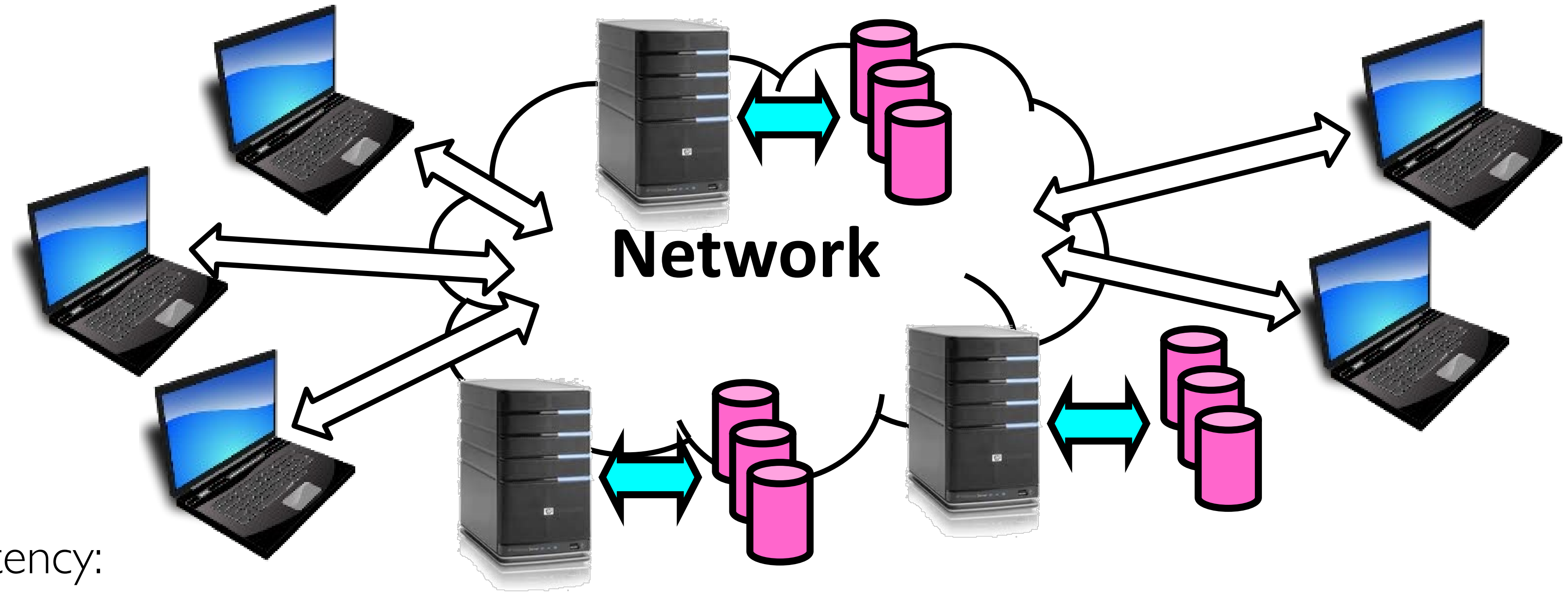
- Example: split kernel into application-level servers.
  - File system looks remote, even though on same machine





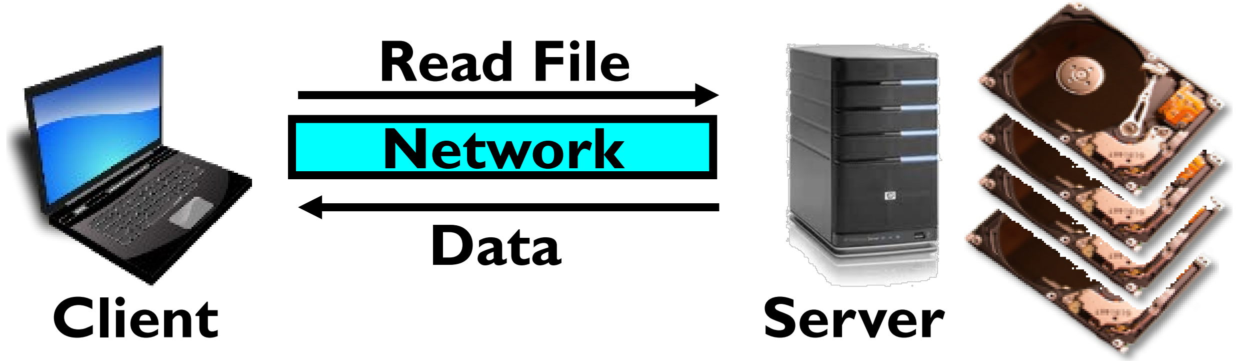
- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - » For example in the X windowing system: Each X client can be on a separate machine from X server; Neither has to run on the machine with the frame buffer.

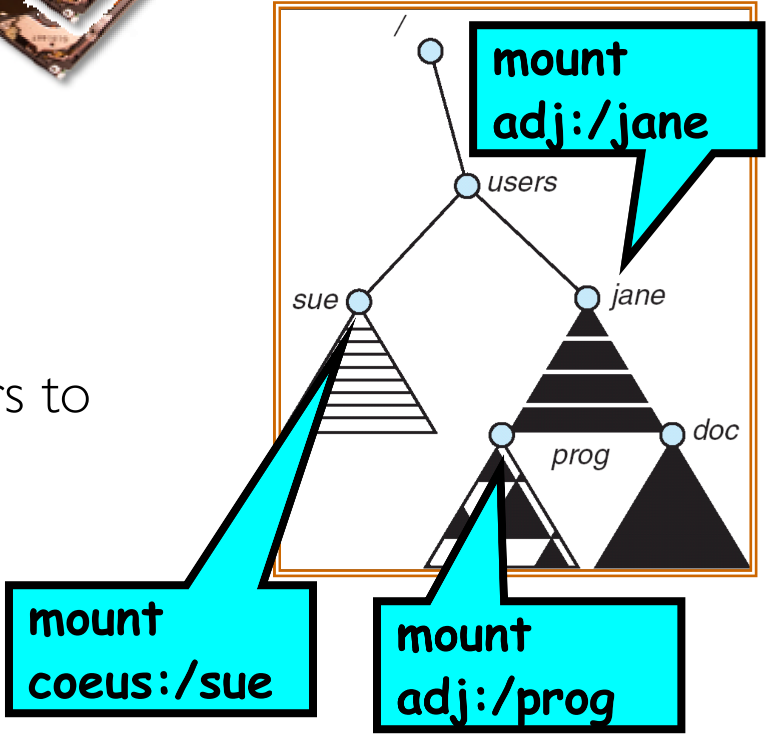
Network-Attached Storage and the CAP Theorem



- Consistency:
  - Changes appear to everyone in the same serial order
- Availability:
  - Can get a result at any time
- Partition-Tolerance
  - System continues to work even when network becomes partitioned
- Consistency, Availability, Partition-Tolerance (CAP) Theorem: Cannot have all three at same time
  - Otherwise known as "Brewer's Theorem"

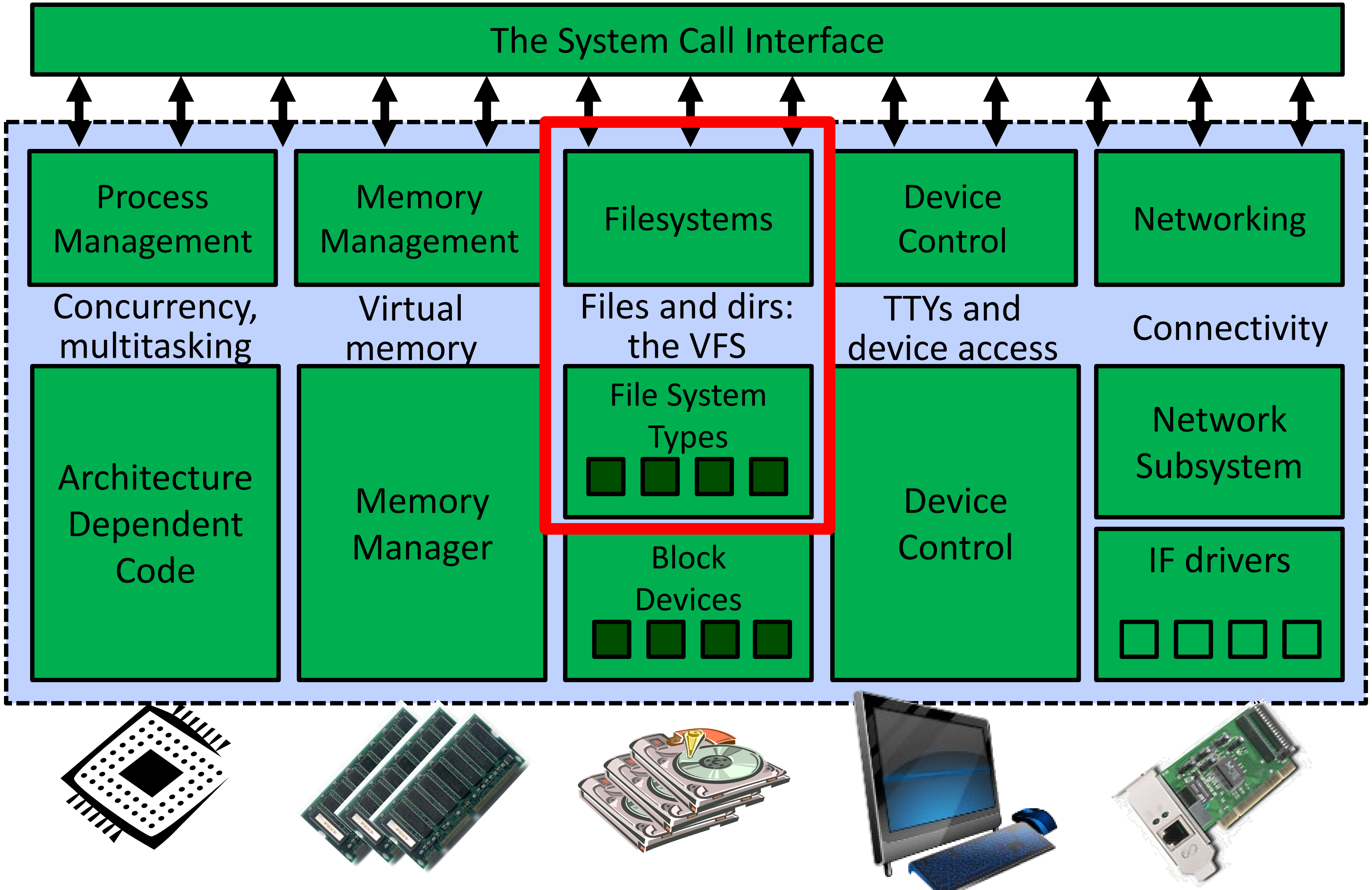
Distributed File Systems



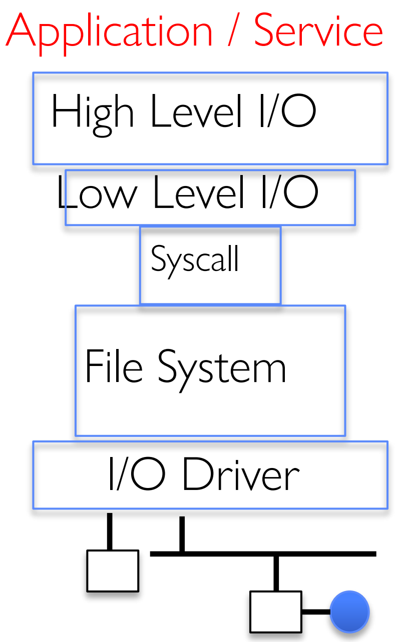


- Transparent access to files stored on a remote disk
- *Mount* remote files into your local file system
  - Directory in local file system refers to remote files
  - e.g., `/users/jane/prog/foo.c` on laptop actually refers to `/prog/foo.c` on `adj.cs.berkeley.edu`
- *Naming* Choices:
  - [Hostname,localname]: Filename includes server
    - » No location or migration transparency, except through DNS remapping
  - A global name space: Filename unique in "world"
    - » Can be served by any server

Enabling Design: VFS



Recall: Layers of I/O…



User App:

```
length = read(input_fd, buffer, BUFFER_SIZE);
```

User library:

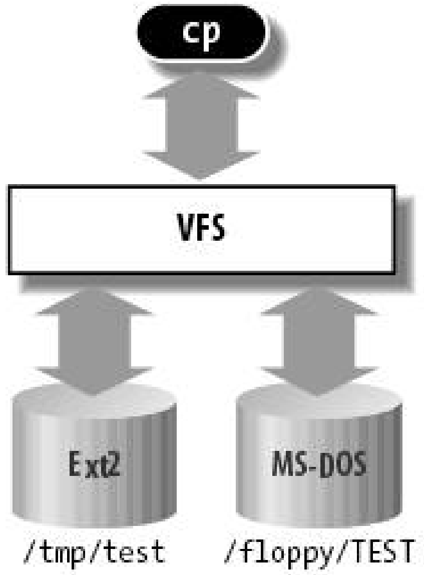
```
ssize_t read(int, void *, size_t) {
  marshal args into registers
  issue syscall
  register result of syscall to rtn value
};
```

```
Exception U→K, interrupt processing
 void syscall_handler (struct intr_frame *f) {
    unmarshall call#, args from regs
    dispatch : handlers[call#](args)
    marshal results fo syscall ret
 }
```

```
ssize_t vfs_read(struct file *file, char __user *buf,
                 size_t count, loff_t *pos) {
   User Process/File System relationship
   call device driver to do the work
}
```

Device Driver

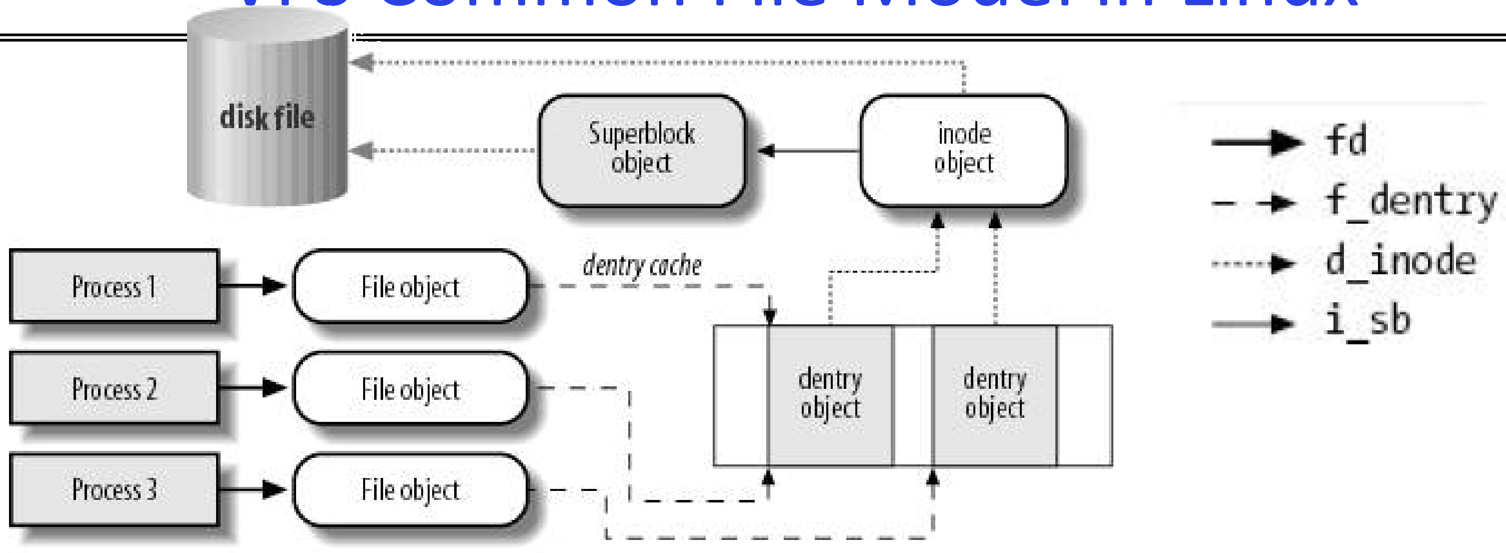
Virtual Filesystem Switch



```
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);
```

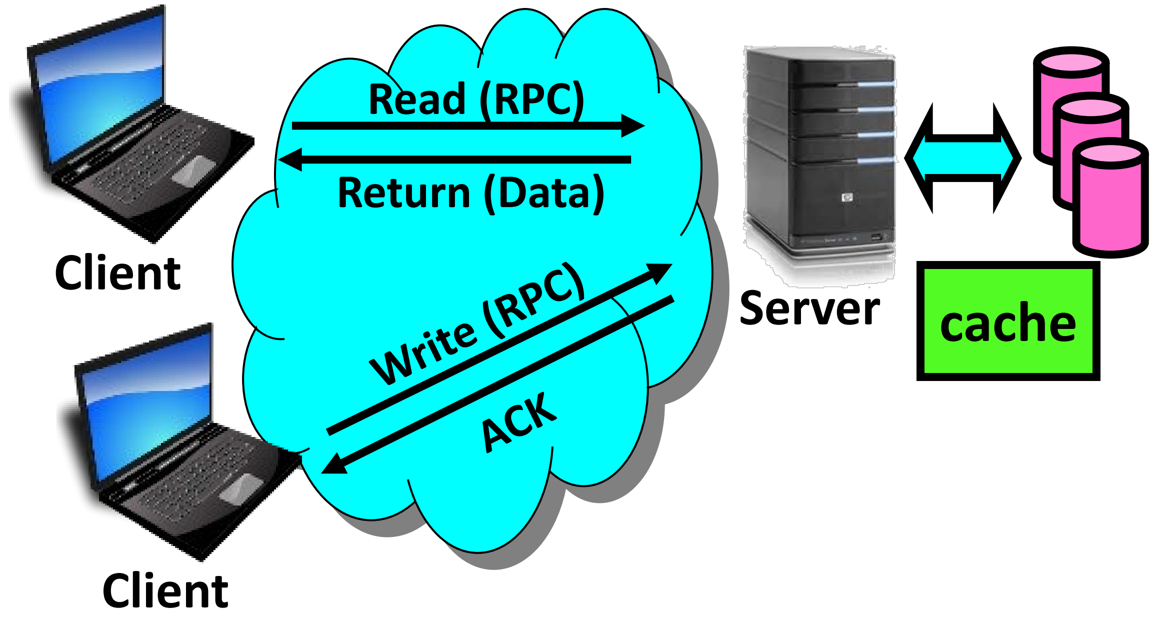
- VFS: Virtual abstraction similar to local file system
  - Provides virtual superblocks, inodes, files, etc
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

VFS Common File Model in Linux



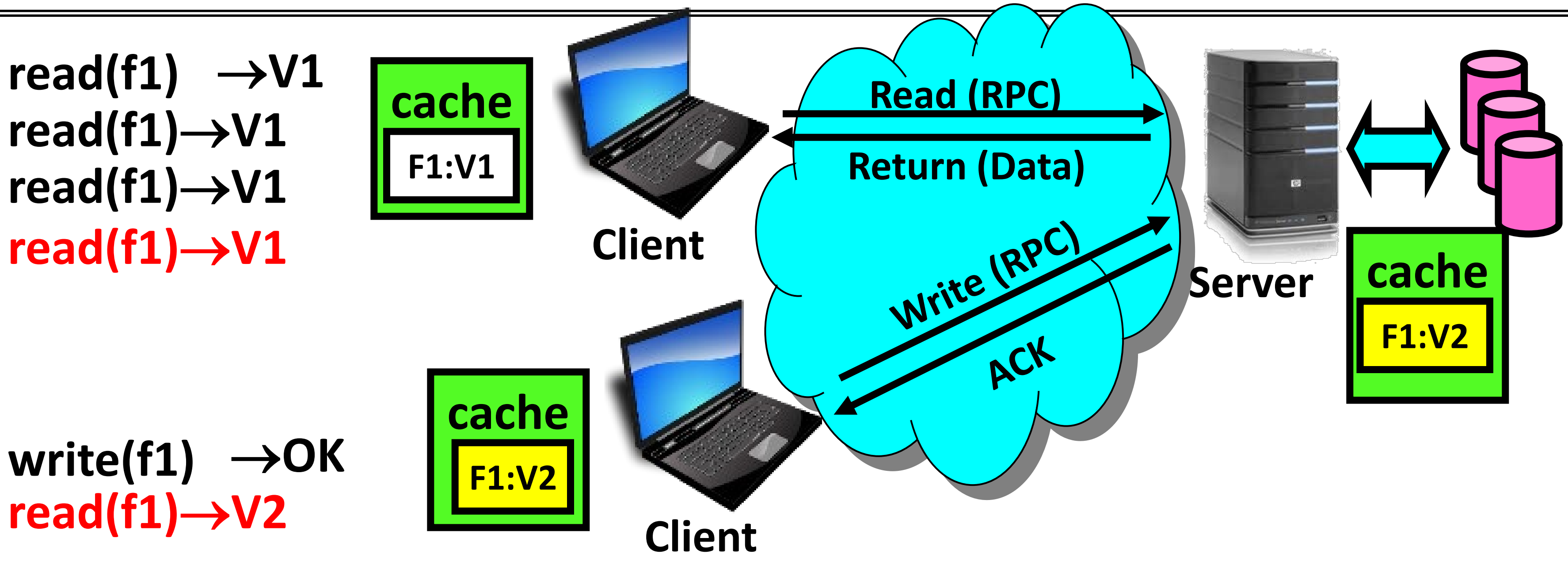
- Four primary object types for VFS:
  - superblock object: represents a specific mounted filesystem
  - inode object: represents a specific file
  - dentry object: represents a directory entry
  - file object: represents open file associated with process
- There is no specific directory object (VFS treats directories as files)
- May need to fit the model by faking it
  - Example: make it look like directories are files
  - Example: make it look like have inodes, superblocks, etc.

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
  - No local caching, but can be cache at server-side
- 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

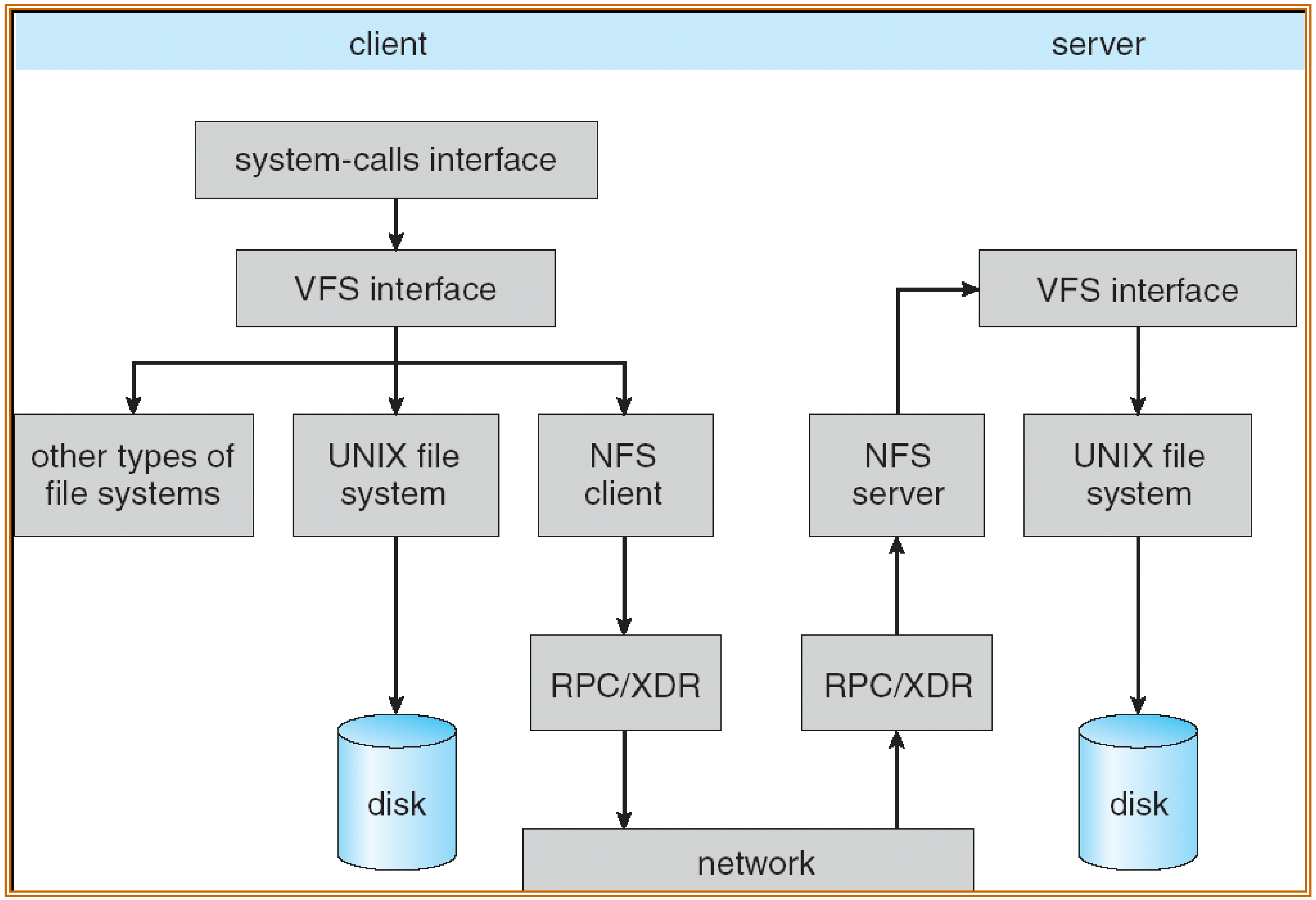
Case Study: Network File System (NFS)

- 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
  - XDR Serialization standard for data format independence
  - 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

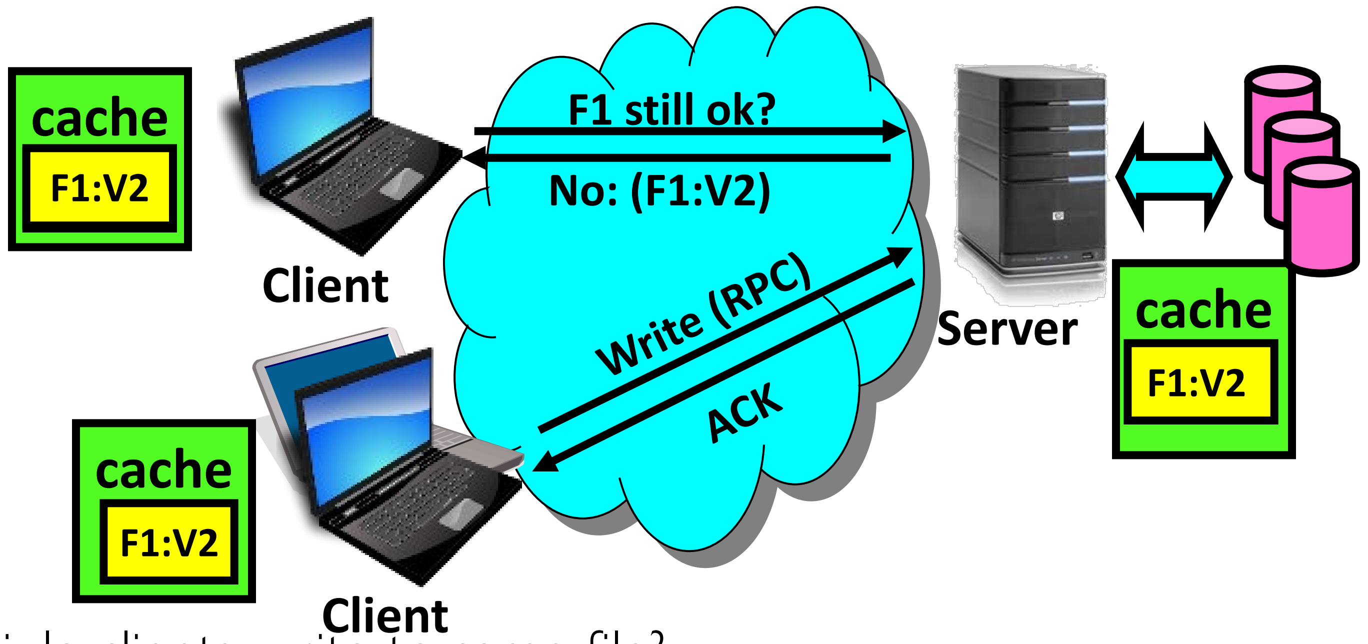
- NFS servers are stateless; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as **`ReadAt(inumber,position)`**, not **`Read(openfile)`**
  - No need to perform network open() or close() on file – each operation stands on its own
- 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

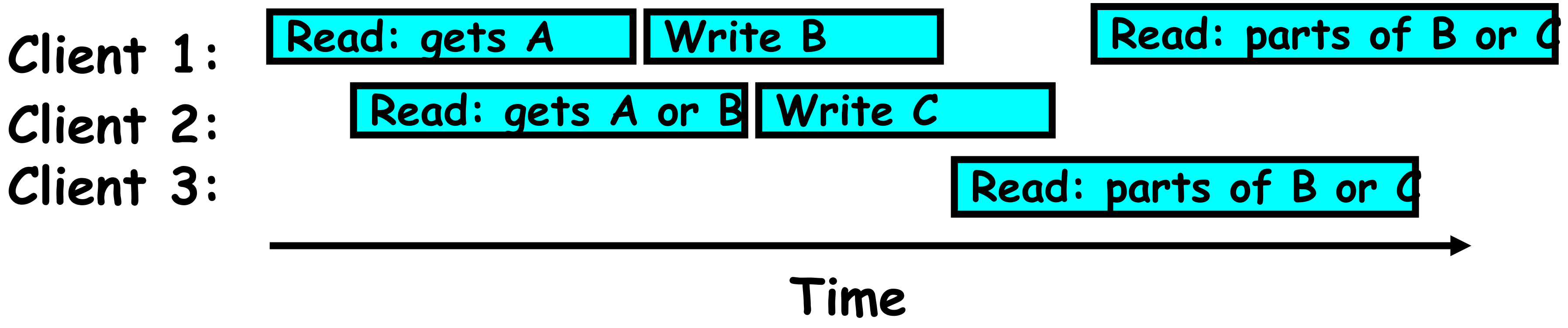
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version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

Andrew File System

- Andrew File System (AFS, late 80's) → DCE DFS (commercial product)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - » As a result, do not get partial writes: all or nothing!
    - » Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

Andrew File System (con't)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - » Get file from server, set up callback with server
  - On write followed by close:
    - » Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"
- AFS Pro: Relative to NFS, less server load:
  - Disk as cache $\Rightarrow$ more files can be cached locally
  - Callbacks $\Rightarrow$ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes$\rightarrow$server, cache misses$\rightarrow$server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost relative to workstation

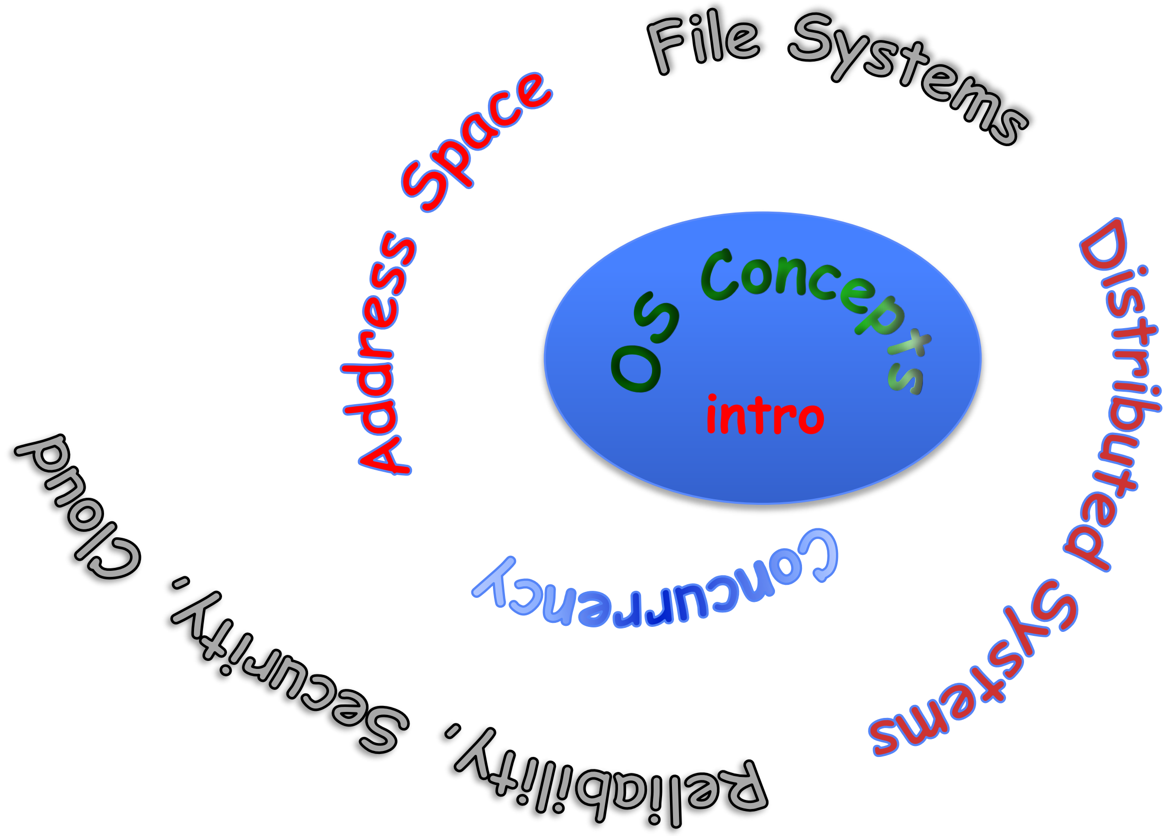
Summary (1/2)

- TCP: Reliable byte stream between two processes on different machines over Internet (read, write, flush)
  - Uses window-based acknowledgement protocol
  - Congestion-avoidance dynamically adapts sender window to account for congestion in network
- Remote Procedure Call (RPC): Call procedure on remote machine or in remote domain
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)
  - Adapts automatically to different hardware and software architectures at remote end

Summary (2/2)

- Distributed File System:
  - Transparent access to files stored on a remote disk
  - Caching for performance
- VFS: Virtual File System layer (Or Virtual Filesystem Switch)
  - Provides mechanism which gives same system call interface for different types of file systems
- Cache Consistency: Keeping client caches consistent with one another
  - If multiple clients, some reading and some writing, how do stale cached copies get updated?
  - NFS: check periodically for changes
  - AFS: clients register callbacks to be notified by server of changes

Thank you!


File Systems
Address Space
OS Concepts
intro
Distributed Systems
Concurrency
Reliability, Security, Cloud