Distributed Systems

Using Messages: Send/Receive behavior

- When should `send(message,mbox)` return?
  - When receiver gets message? (i.e. ack received)
  - When message is safely buffered on destination?
  - Right away, if message is buffered on source node?
- Actually two questions here:
  - When can the sender be sure that receiver actually received the message?
  - When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from T1→T2
  - T1→buffer→T2
  - Very similar to producer/consumer
    - Send = V, Receive = P
    - However, can't tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

```
Producer:
   int msg1[1000];
   while(1) {
      prepare message;
      send(msg1,mbox);
   }
Consumer:
   int buffer[1000];
   while(1) {
      receive(buffer,mbox);
      process message;
   }
```

Send Message

Receive Message

- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  - One of the roles of the window in TCP: window is size of buffer on far end
  - Restricts sender to forward only what will fit in buffer

Messaging for Request/Response communication

- What about two-way communication?
  - Request/Response
    - Read a file stored on a remote machine
    - Request a web page from a remote web server
  - Also called: client-server
    - Client ≡ requester, Server ≡ responder
    - Server provides "service" (file storage) to the client
- Example: File service

```
Client: (requesting the file)
   char response[1000];

   send("read rutabaga", server_mbox);
   receive(response, client_mbox);

Server: (responding with the file)
   char command[1000], answer[1000];

   receive(command, server_mbox);
   decode command;
   read file into answer;
   send(answer, client_mbox);
```

Request File

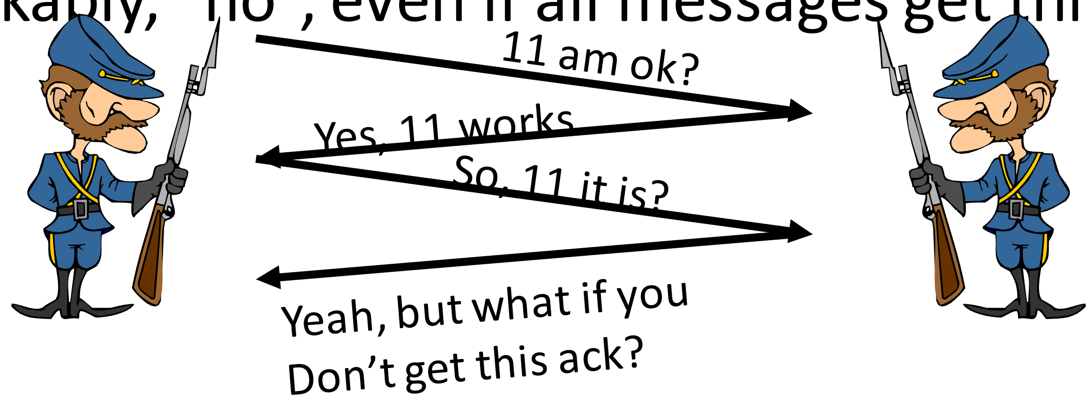
Get Response

Receive Request

Send Response

General's Paradox

- General's paradox:
  - Constraints of problem:
    - Two generals, on separate mountains
    - Can only communicate via messengers
    - Messengers can be captured
  - Problem: need to coordinate attack
    - If they attack at different times, they all die
    - If they attack at same time, they win
  - Named after Custer, who died at Little Big Horn because he arrived a couple of days too early
- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, "no", even if all messages get through



  - No way to be sure last message gets through!

Two-Phase Commit

- Since we can't solve the General's Paradox (i.e. simultaneous action), let's solve a related problem
  - Distributed transaction: Two machines agree to do something, or not do it, atomically
- Two-Phase Commit protocol does this
  - Use a persistent, stable log on each machine to keep track of whether commit has happened
    - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
  - Prepare Phase:
    - The global coordinator requests that all participants will promise to commit or rollback the transaction
    - Participants record promise in log, then acknowledge
    - If anyone votes to abort, coordinator writes "`Abort`" in its log and tells everyone to abort; each records "`Abort`" in log
  - Commit Phase:
    - After all participants respond that they are prepared, then the coordinator writes "`Commit`" to its log
    - Then asks all nodes to commit; they respond with ack
    - After receive acks, coordinator writes "`Got Commit`" to log
  - Log can be used to complete this process such that all machines either commit or don't commit

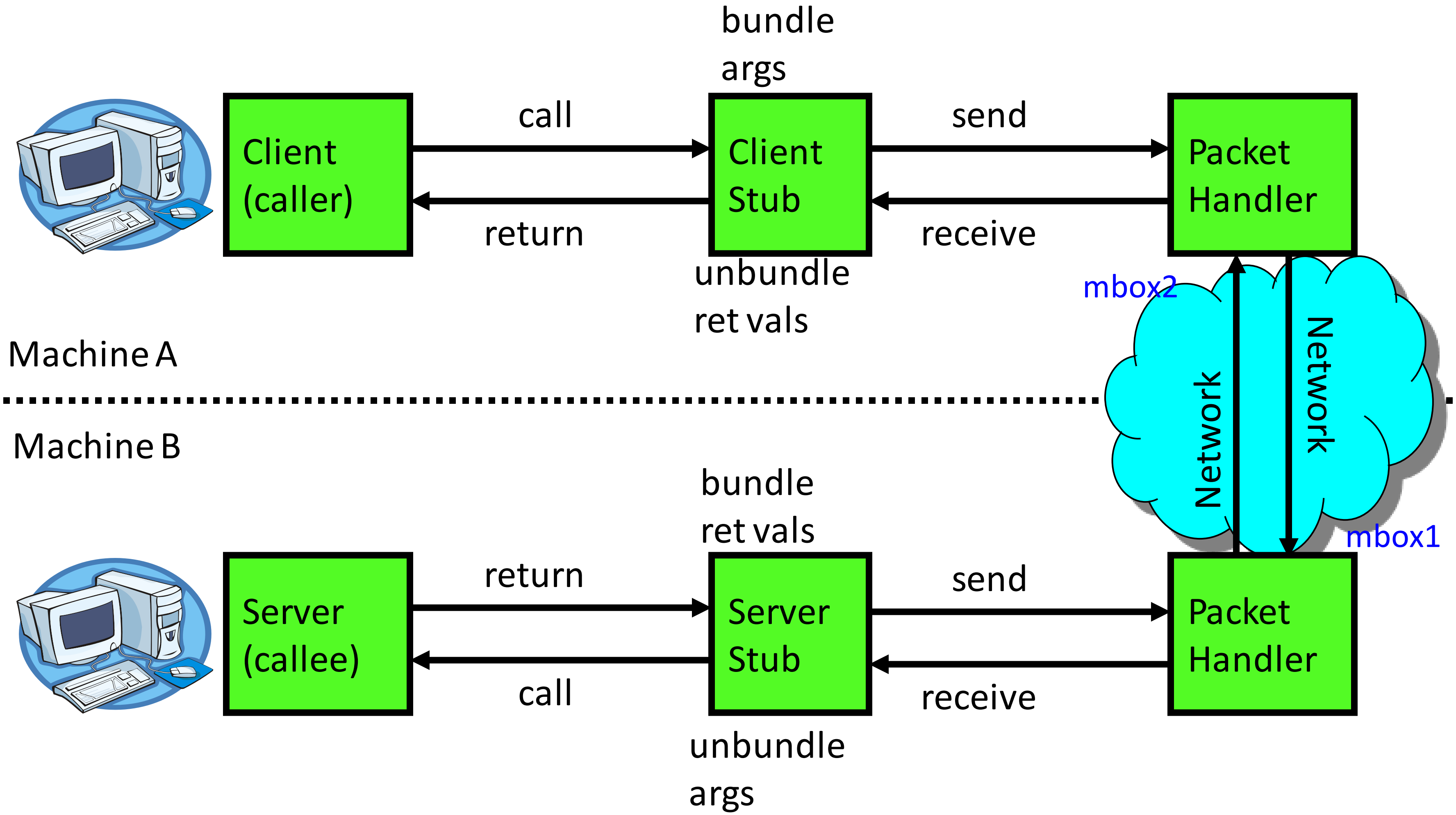
Two phase commit example

- Simple Example: A≡Canara Bank, B≡SBI
  - Phase 1: Prepare Phase
    - A writes "`Begin transaction`" to log
      A→B: OK to transfer funds to me?
    - Not enough funds:
      B→A: transaction aborted; A writes "`Abort`" to log
    - Enough funds:
      B: Write new account balance & promise to commit to log
      B→A: OK, I can commit
  - Phase 2: A can decide for both whether they will commit
    - A: write new account balance to log
    - Write "`Commit`" to log
    - Send message to B that commit occurred; wait for ack
    - Write "`Got Commit`" to log
- What if B crashes at beginning?
  - Wakes up, does nothing; A will timeout, abort and retry
- What if A crashes at beginning of phase 2?
  - Wakes up, sees that there is a transaction in progress; sends "`Abort`" to B
- What if B crashes at beginning of phase 2?
  - B comes back up, looks at log; when A sends it "`Commit`" message, it will say, "oh, ok, commit"

Remote Procedure Call

- 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
- Better option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    `remoteFileSystem→Read("rutabaga");`
  - Translated automatically into call on server:
    `fileSys→Read("rutabaga");`
- 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 Information Flow



RPC Details

- 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
- Cross-platform issues:
  - What if client/server machines are different architectures or in different languages?
    - Convert everything to/from some canonical form
    - Tag every item with an indication of how it is encoded (avoids unnecessary conversions).

RPC Details (continued)

- How does client know which mbox 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
- 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

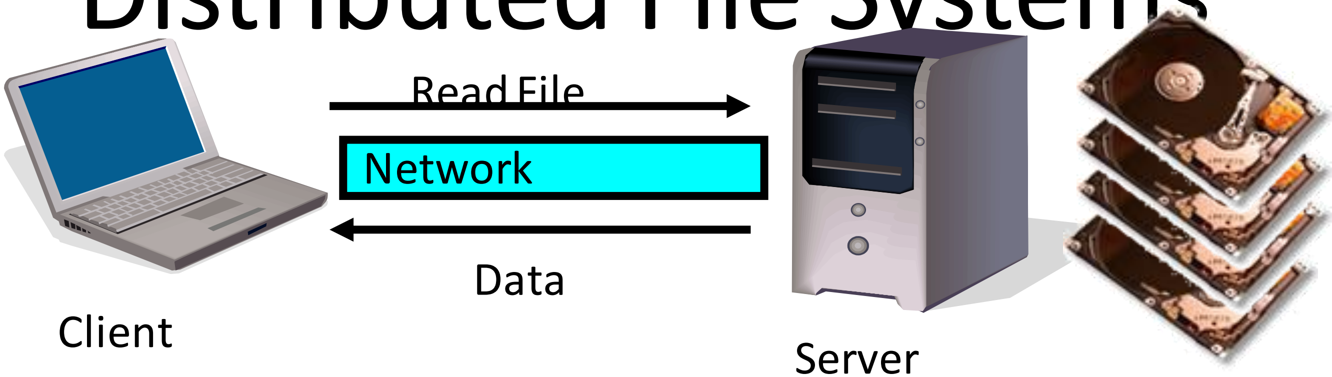
Problems with RPC

- Non-Atomic failures
  – Different failure modes in distributed 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
- Performance
  – Cost of Procedure call « same-machine RPC « network RPC
  – Means programmers must be aware that RPC is not free
    - Caching can help, but may make failure handling complex

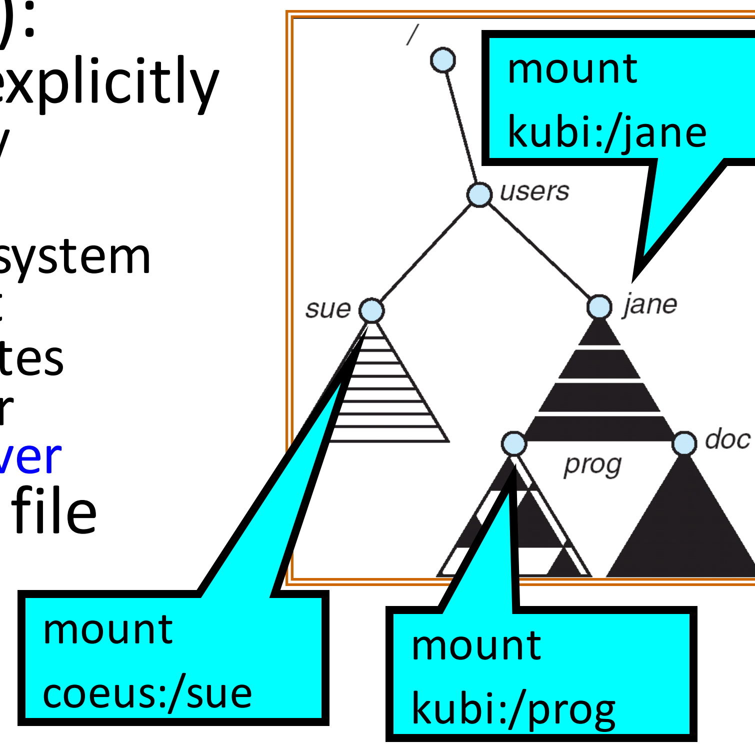
Cross-Domain Communication/Location Transparency

- 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 modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)

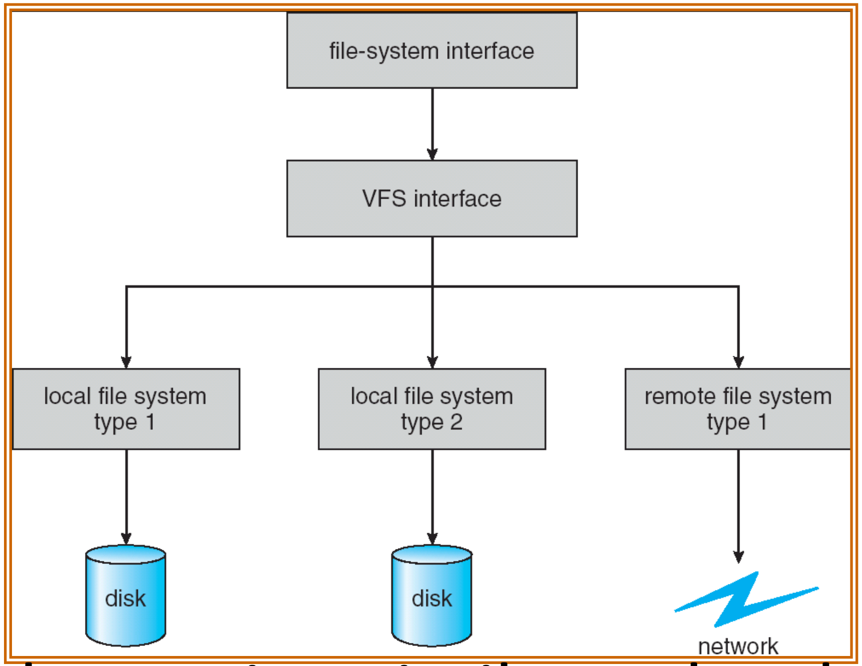
Distributed File Systems



- Distributed File System:
  - Transparent access to files stored on a remote disk
- Naming choices (always an issue):
  - *Hostname:*localname: Name files explicitly
    - No location or migration transparency
  - *Mounting* of remote file systems
    - System manager mounts remote file system by giving name and local mount point
    - Transparent to user: all reads and writes look like local reads and writes to user e.g. /users/sue/foo→/sue/foo on server
  - *A single, global name space:* every file in the world has unique name
    - Location Transparency: servers can change and files can move without involving user

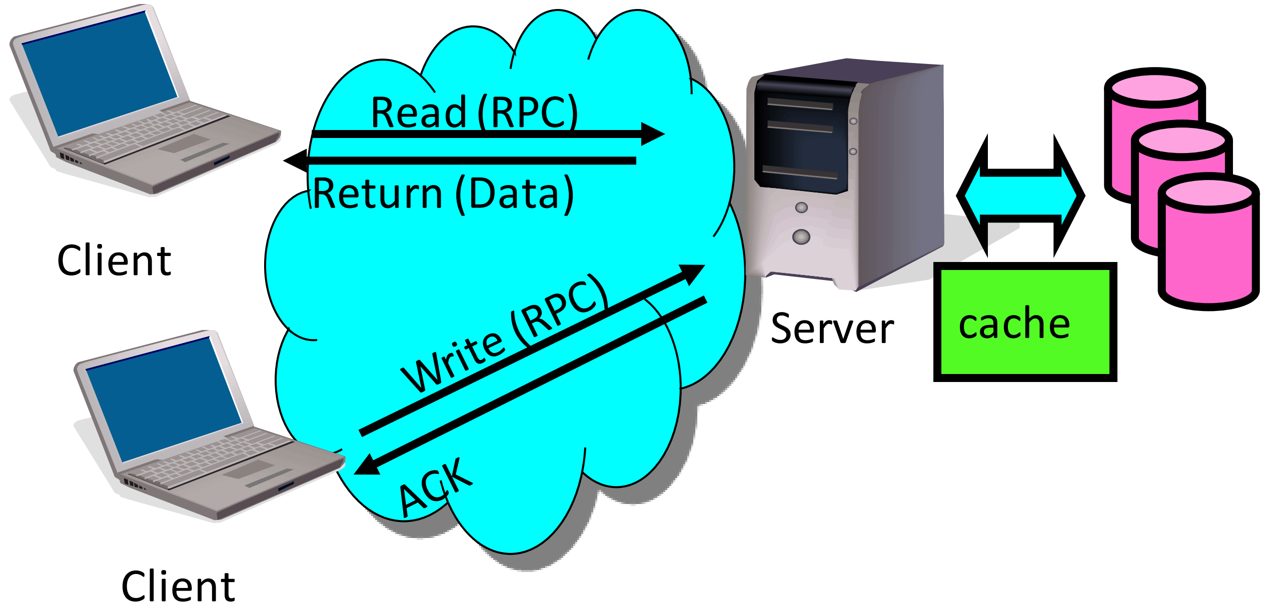


Virtual File System (VFS)



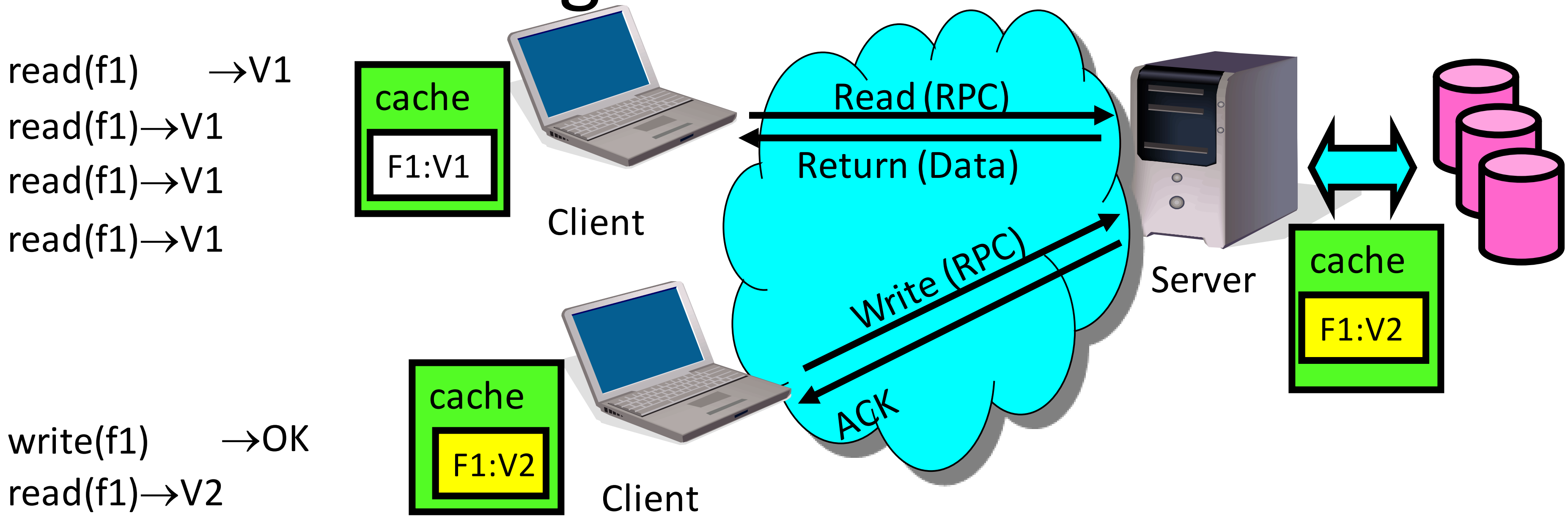
- VFS: Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
  - Compatible with a variety of local and remote file systems
    - provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types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

Simple Distributed File System



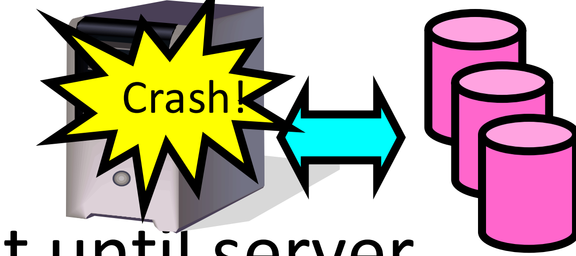
- Remote Disk: Reads and writes forwarded to server
  - Use RPC to translate file system calls
  - No local caching/can be caching at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going 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

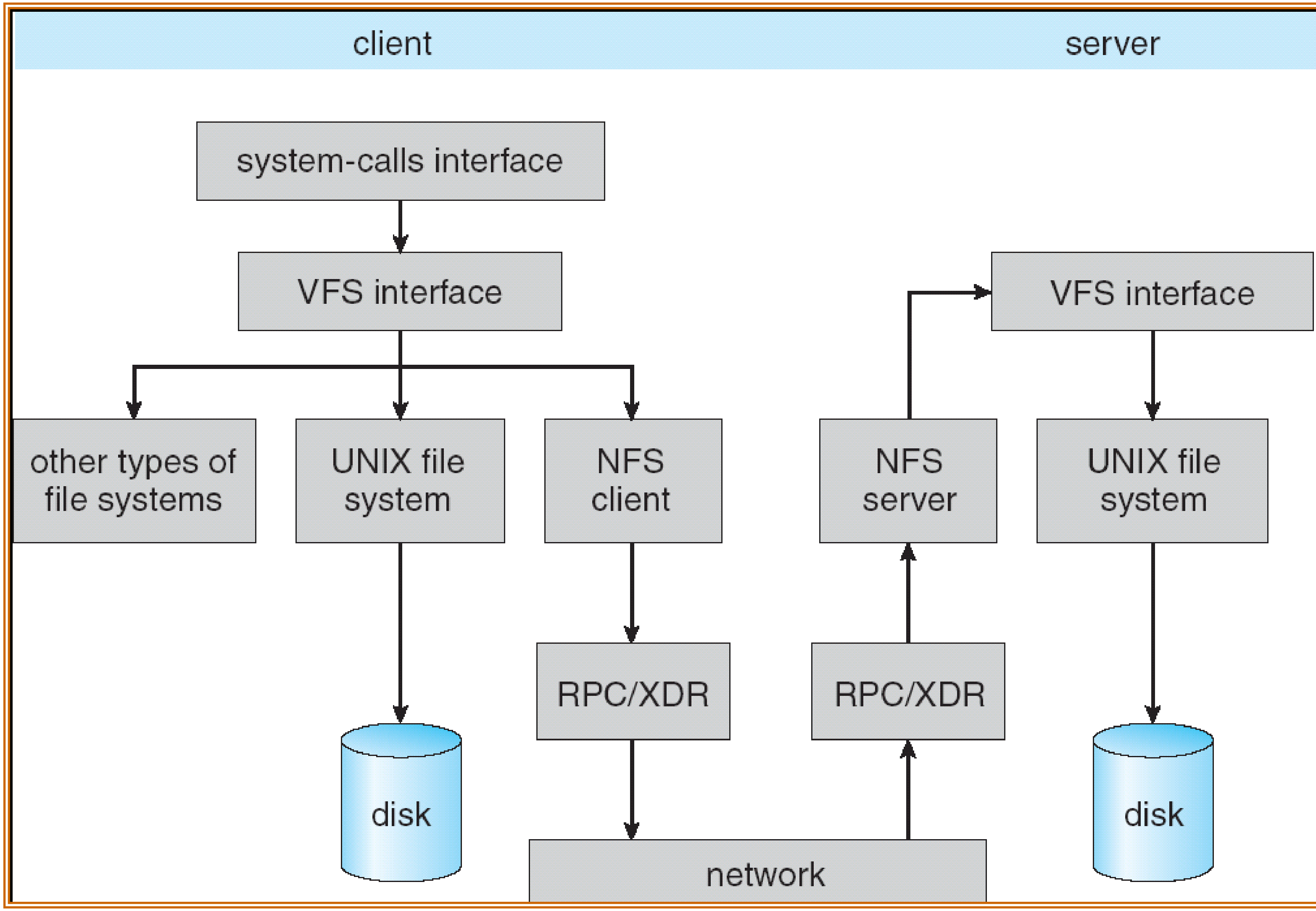
Failures



- What if server crashes? Can client wait until server comes back up and continue as before?
  - Any data in server memory but not on disk can be lost
  - Shared state across RPC: What if server crashes after seek? Then, when client does "read", it will fail
  - Message retries: suppose server crashes after it does UNIX "rm foo", but before acknowledgment?
    - Message system will retry: send it again
    - How does it know not to delete it again? (could solve with two-phase commit protocol, but NFS takes a more ad hoc approach)

- Stateless protocol: A protocol in which all information required to process a request is passed with request
  - Server keeps no state about client, except as hints to help improve performance (e.g. a cache)
  - Thus, if server crashes and restarted, requests can continue where left off (in many cases)
- What if client crashes?
  - Might lose modified data in client cache

Schematic View of NFS Architecture



Network File System (NFS)

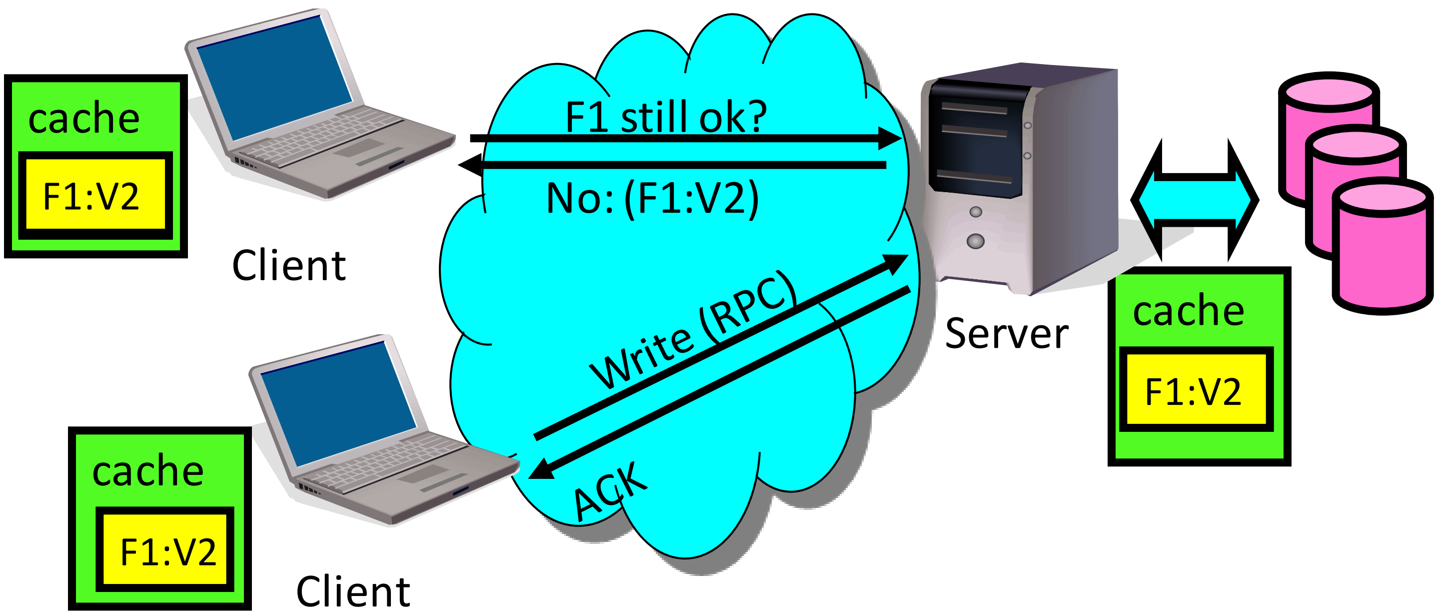
- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - Implements the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

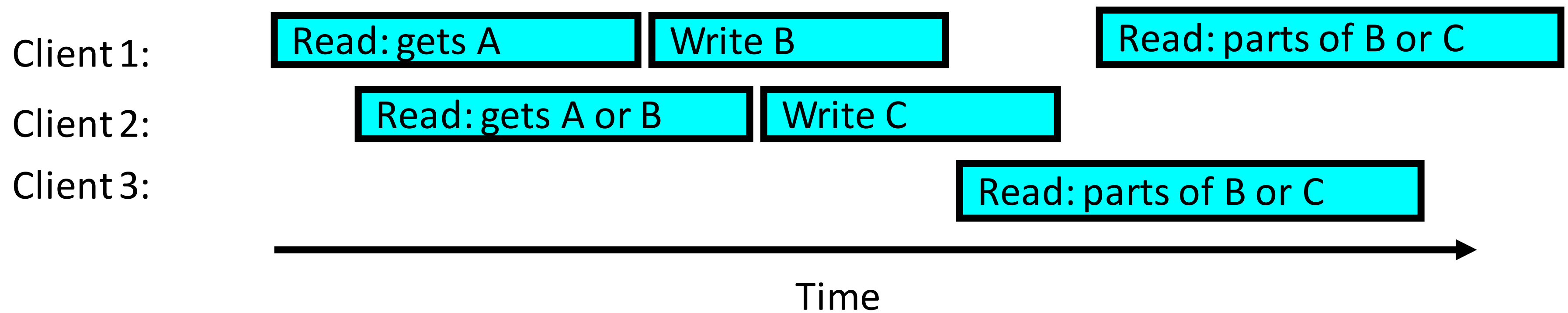
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - Thus, when file is changed on one client, server is notified, but other clients use old version of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

World Wide Web

- Key idea: graphical front-end to RPC protocol
- What happens when a web server fails?
  - System breaks!
  - Solution: Transport or network-layer redirection
    - Invisible to applications
    - Can also help with scalability (load balancers)
    - Must handle “sessions” (e.g., banking/e-commerce)
- Initial version: no caching
  - Didn’t scale well – easy to overload servers

WWW Caching

- Use client-side caching to reduce number of interactions between clients and servers and/or reduce the size of the interactions:
  - Time-to-Live (TTL) fields – HTTP “Expires” header from server
  - Client polling – HTTP “If-Modified-Since” request headers from clients
  - Server refresh – HTML “META Refresh tag” causes periodic client poll
- What is the polling frequency for clients and servers?
  - Could be adaptive based upon a page’s age and its rate of change
- Server load is still significant!

WWW Proxy Caches

- Place caches in the network to reduce server load
  - But, increases latency in lightly loaded case
  - Caches near servers called "reverse proxy caches"
    - Offloads busy server machines
  - Caches at the "edges" of the network called "content distribution networks"
    - Offloads servers and reduce client latency
- Challenges:
  - Caching static traffic easy, but only ~40% of traffic
  - Dynamic and multimedia is harder
    - Multimedia is a big win: Megabytes versus Kilobytes
  - Same cache consistency problems as before
- Caching is changing the Internet architecture
  - Places functionality at higher levels of comm. protocols

Conclusion

- Remote Procedure Call (RPC): Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)
- VFS: Virtual File System layer
  - Provides mechanism which gives same system call interface for different types of file systems
- Distributed File System:
  - Transparent access to files stored on a remote disk
    - NFS: Network File System
    - AFS: Andrew File System
  - Caching for performance
- Cache Consistency: Keeping contents of 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 so can be notified by server of changes