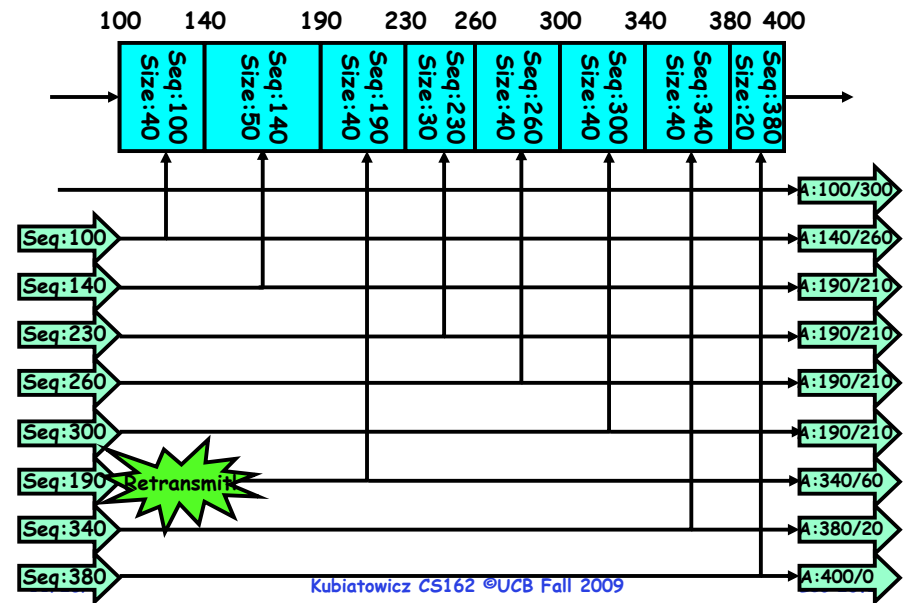


CS162
Operating Systems and
Systems Programming
Lecture 23

Network Communication Abstractions /
Distributed Programming

November 25, 2009
Prof. John Kubiatowicz
<http://inst.eecs.berkeley.edu/~cs162>

Review: Window-Based Acknowledgements (TCP)



Review: Congestion Avoidance

- Congestion
 - How long should timeout be for re-sending messages?
 - » Too long → wastes time if message lost
 - » Too short → retransmit even though ack will arrive shortly
 - Stability problem: more congestion ⇒ ack is delayed ⇒ unnecessary timeout ⇒ more traffic ⇒ 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 ⇒ congestion, so cut window size in half
 - "Additive Increase, Multiplicative Decrease"

Goals for Today

- Finish Discussion of TCP/IP
- Messages
 - Send/receive
 - One vs. two-way communication
- Distributed Decision Making
 - Two-phase commit/Byzantine Commit
- Remote Procedure Call
- Distributed File Systems (Part I)

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Sequence-Number Initialization

- How do you choose an initial sequence number?
 - When machine boots, ok to start with sequence #0?
 - » No: could send two messages with same sequence #!
 - » Receiver might end up discarding valid packets, or duplicate ack from original transmission might hide lost packet
 - Also, if it is possible to predict sequence numbers, might be possible for attacker to hijack TCP connection
- Some ways of choosing an initial sequence number:
 - Time to live: each packet has a deadline.
 - » If not delivered in X seconds, then is dropped
 - » Thus, can re-use sequence numbers if wait for all packets in flight to be delivered or to expire
 - Epoch #: uniquely identifies *which* set of sequence numbers are currently being used
 - » Epoch # stored on disk, Put in every message
 - » Epoch # incremented on crash and/or when run out of sequence #
 - Pseudo-random increment to previous sequence number
 - » Used by several protocol implementations

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.5

Use of TCP: Sockets

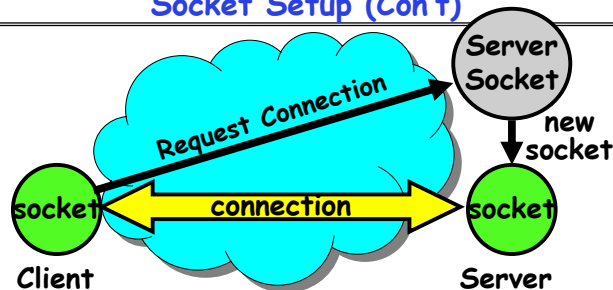
- **Socket:** an abstraction of a network I/O queue
 - Embodies one side of a communication channel
 - » Same interface regardless of location of other end
 - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
 - First introduced in 4.2 BSD UNIX: big innovation at time
 - » Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
 - On server: set up "server-socket"
 - » Create socket, Bind to protocol (TCP), local address, port
 - » Call listen(): tells server socket to accept incoming requests
 - » Perform multiple accept() calls on socket to accept incoming connection request
 - » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
 - On client:
 - » Create socket, Bind to protocol (TCP), remote address, port
 - » Perform connect() on socket to make connection
 - » If connect() successful, have socket connected to server

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.6

Socket Setup (Con't)



- Things to remember:
 - Connection involves 5 values:
[Client Addr, Client Port, Server Addr, Server Port, Protocol]
 - 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
- Note that the uniqueness of the tuple is really about two Addr/Port pairs and a protocol

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.7

Socket Example (Java)

```
server:
//Makes socket, binds addr/port, calls listen()
ServerSocket sock = new ServerSocket(6013);
while(true) {
    Socket client = sock.accept();
    PrintWriter pout = new
        PrintWriter(client.getOutputStream(),true);

    pout.println("Here is data sent to client!");
    ...
    client.close();
}

client:
// Makes socket, binds addr/port, calls connect()
Socket sock = new Socket("169.229.60.38",6013);
BufferedReader bin =
    new BufferedReader(
        new InputStreamReader(sock.getInputStream()));
String line;
while ((line = bin.readLine())!=null)
    System.out.println(line);
sock.close();
```

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.8

Distributed Applications

- 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

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.9

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!

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.10

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

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

Send Message

```
Consumer:
int buffer[1000];
while(1) {
    receive(buffer, mbox);
    process 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

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.11

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);
```

Request File

Get Response

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

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

Receive Request

Send Response

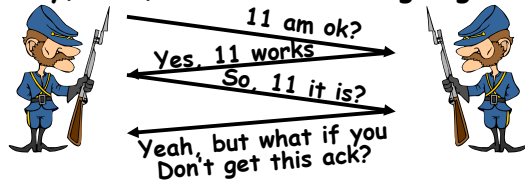
11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.12

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!

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.13

Administrivia

- Final Exam
 - Thursday 12/17, 8:00AM-11:00AM, 105 Stanley Hall
 - All material from the course
 - » With slightly more focus on second half, but you are still responsible for all the material
 - Two sheets of notes, both sides
 - Will need dumb calculator
- Should be working on Project 4
 - Last one!
- There *is* a lecture on Wednesday before Thanksgiving
 - Including this one, we are down to 6 lectures...!
 - Upside: You get extra week of study before finals

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.14

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

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.15

Two phase commit example

- Simple Example: A≡WellsFargo Bank, B≡Bank of America
 - 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"

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.16

Distributed Decision Making Discussion

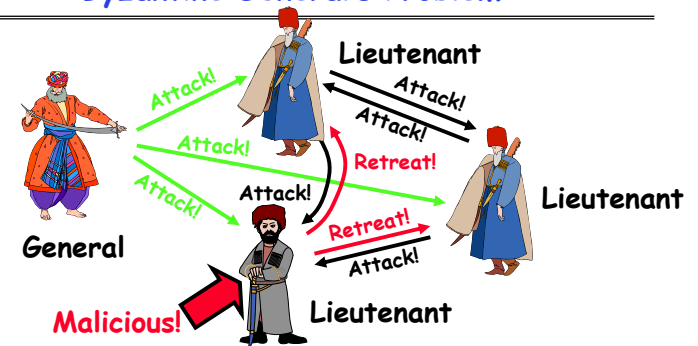
- Why is distributed decision making desirable?
 - Fault Tolerance!
 - A group of machines can come to a decision even if one or more of them fail during the process
 - » Simple failure mode called "failstop" (different modes later)
 - After decision made, result recorded in multiple places
- Undesirable feature of Two-Phase Commit: Blocking
 - One machine can be stalled until another site recovers:
 - » Site B writes "prepared to commit" record to its log, sends a "yes" vote to the coordinator (site A) and crashes
 - » Site A crashes
 - » Site B wakes up, check its log, and realizes that it has voted "yes" on the update. It sends a message to site A asking what happened. At this point, B cannot decide to abort, because update may have committed
 - » B is blocked until A comes back
 - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update
- Alternative: There are alternatives such as "Three Phase Commit" which don't have this blocking problem
- What happens if one or more of the nodes is malicious?
 - **Malicious:** attempting to compromise the decision making

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.17

Byzantine General's Problem



- Byzantine General's Problem (n players):
 - One General
 - n-1 Lieutenants
 - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that:
 - IC1: All loyal lieutenants obey the same order
 - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

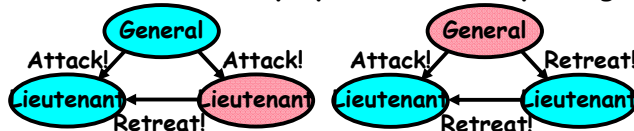
11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

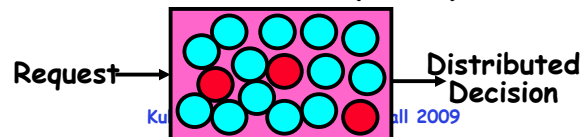
Lec 23.18

Byzantine General's Problem (con't)

- Impossibility Results:
 - Cannot solve Byzantine General's Problem with n=3 because one malicious player can mess up things



- With f faults, need $n > 3f$ to solve problem
- Various algorithms exist to solve problem
 - Original algorithm has #messages exponential in n
 - Newer algorithms have message complexity $O(n^2)$
 - » One from MIT, for instance (Castro and Liskov, 1999)
- Use of BFT (Byzantine Fault Tolerance) algorithm
 - Allow multiple machines to make a coordinated decision even if some subset of them ($< n/3$) are malicious



11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.19

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:

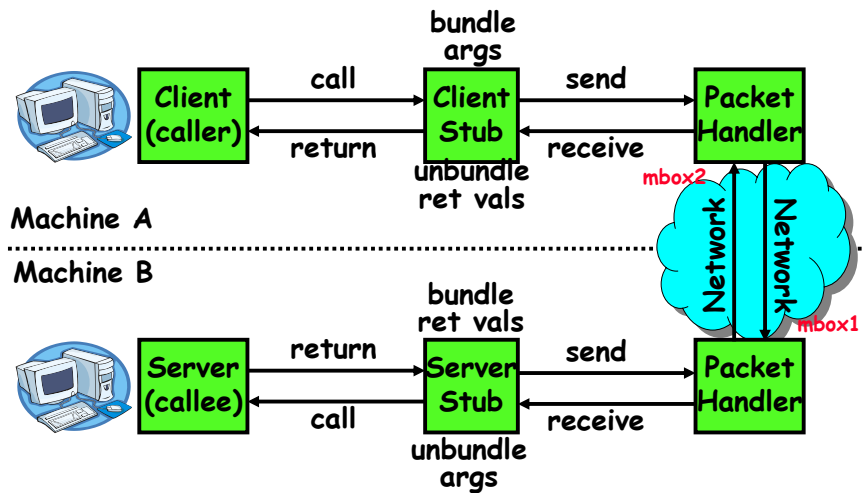

```
fileSys→Read("rutabaga");
```
- Implementation:
 - Request-response message passing (under covers!)
 - "Stub" provides glue on client/server
 - » Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
 - » Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.
- **Marshalling** involves (depending on system)
 - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.20

RPC Information Flow



11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.21

RPC Details

- **Equivalence with regular procedure call**
 - Parameters \leftrightarrow Request Message
 - Result \leftrightarrow 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).

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.22

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 \rightarrow 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

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.23

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 \ll same-machine RPC \ll network RPC
 - Means programmers must be aware that RPC is not free
 - » Caching can help, but may make failure handling complex

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.24

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)

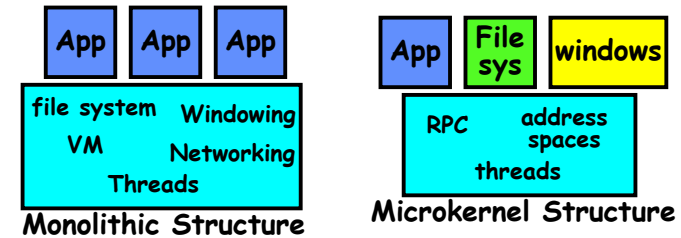
11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.25

Microkernel operating systems

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



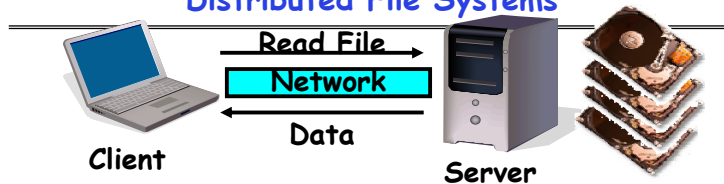
- Why split the OS into separate domains?
 - Fault isolation: bugs are more isolated (build a firewall)
 - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
 - Location transparent: service can be local or remote
 - » For example in the X windowing system: Each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

11/25/09

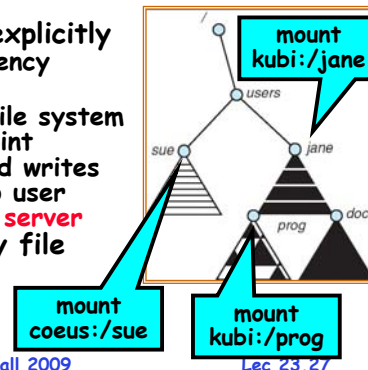
Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.26

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

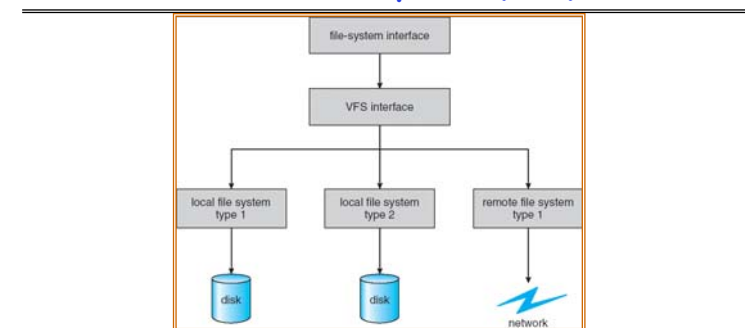


11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.27

Virtual File System (VFS)



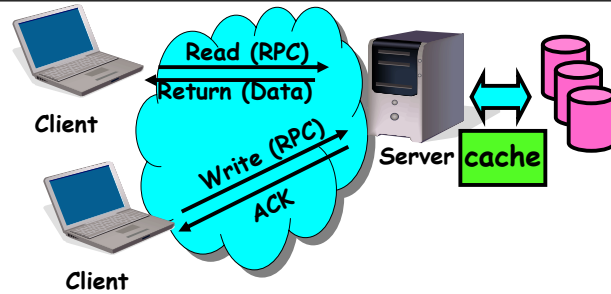
- VFS: Virtual abstraction similar to local file system
 - Instead of "inodes" has "vnodes"
 - Compatible with a variety of local and remote file systems
 - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types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

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.28

Simple Distributed File System



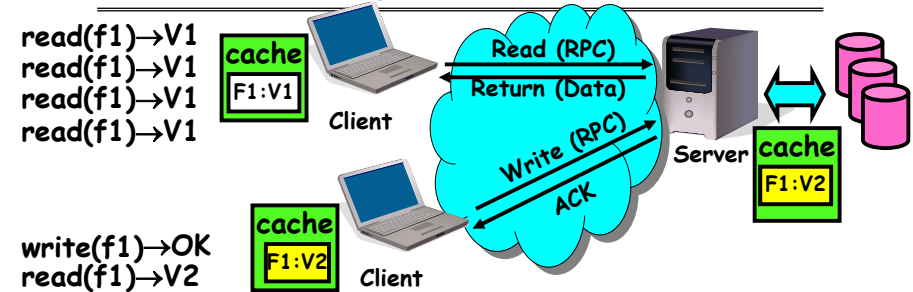
- Remote Disk: Reads and writes forwarded to server
 - Use RPC to translate file system calls
 - No local caching/can be caching at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
 - Going over network is slower than going to local memory
 - Lots of network traffic/not well pipelined
 - Server can be a bottleneck

11/25/09

Kubiatawicz CS162 ©UCB Fall 2009

Lec 23.29

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

11/25/09

Kubiatawicz CS162 ©UCB Fall 2009

Lec 23.30

Failures



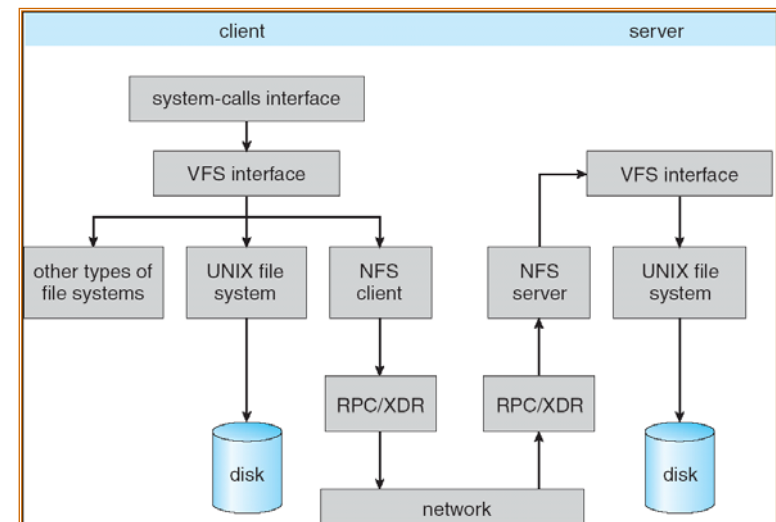
- What if server crashes? Can client wait until server comes back up and continue as before?
 - Any data in server memory but not on disk can be lost
 - Shared state across RPC: What if server crashes after seek? Then, when client does "read", it will fail
 - Message retries: suppose server crashes after 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

11/25/09

Kubiatawicz CS162 ©UCB Fall 2009

Lec 23.31

Schematic View of NFS Architecture



11/25/09

Kubiatawicz CS162 ©UCB Fall 2009

Lec 23.32

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)

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.33

NFS Continued

- NFS servers are **stateless**; each request provides all arguments required 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)

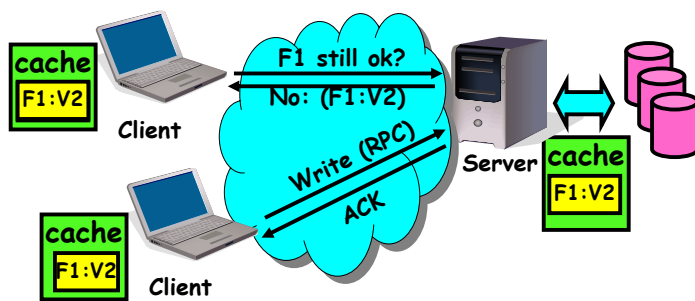
11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.34

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 is 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!

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.35

Conclusion

- **Two-phase commit**: distributed decision making
 - First, make sure everyone guarantees that they will commit if asked (prepare)
 - Next, ask everyone to commit
- **Byzantine General's Problem**: distributed decision making with malicious failures
 - One general, $n-1$ lieutenants: some number of them may be malicious (often "f" of them)
 - All non-malicious lieutenants must come to same decision
 - If general not malicious, lieutenants must follow general
 - Only solvable if $n \geq 3f+1$
- **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
 - Caching for performance

11/25/09

Kubiatowicz CS162 ©UCB Fall 2009

Lec 23.36