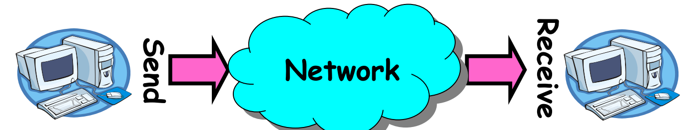
# CS162
# Operating Systems and Systems Programming
# Lecture 24

## Testing Methodologies/ Distributed File Systems

November 22, 2006

Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

## Review: Distributed Applications

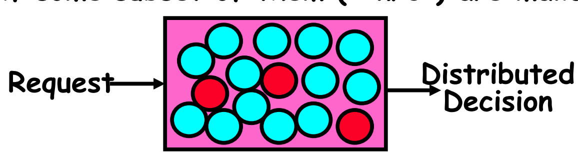



- Message Abstraction: send/receive messages
  - Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (`mbox`): temporary holding area for messages
    - » Includes both destination location and queue
  - `Send(message,mbox)`
    - » 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
- Two-phase commit: distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit



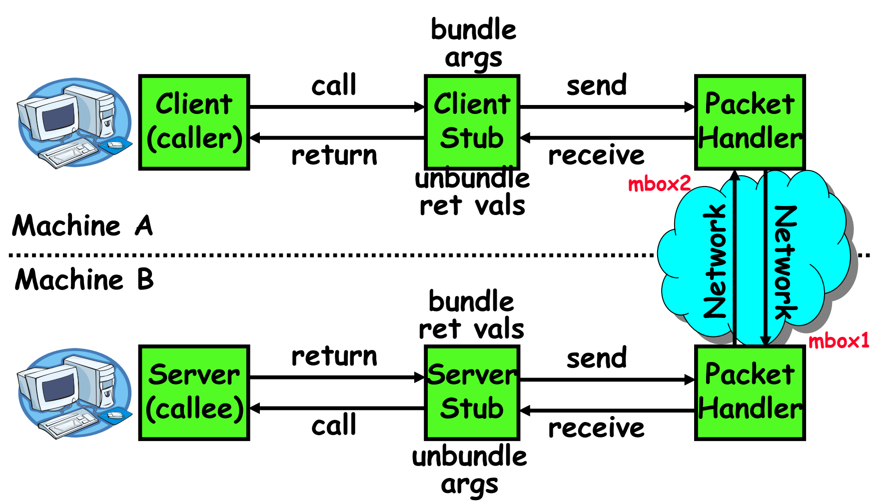
## Review: Byzantine General's Problem

- Byazantine General's Problem (n players):
  - One General
  - n-1 Lieutenants
  - Some number of these (f<n/3) 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
- Various algorithms exist to solve problem
  - Newer algorithms have message complexity $O(n^2)$
- 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






## Review: RPC Information Flow



## Goals for Today

- Finish RPC
- Testing Methodologies
- Examples of Distributed File Systems
- Cache Coherence Protocols

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides on Testing from George Necula (CS169)
Many slides generated from my lecture notes by Kubiatowicz.



## 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 dynmaic 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

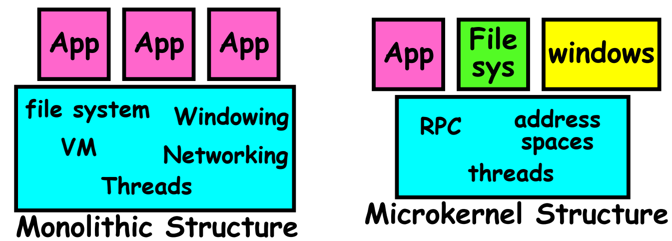
## Cross-Domain Communication/Location Transparency

- 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 on 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)



## 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.

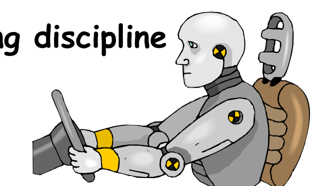


## Administrivia

- My office hours
  - No office hours Thursday (Thanksgiving)
- Project 4 design document
  - Due Tuesday November $28^{th}$
- MIDTERM II: Monday December $4^{th}$!
  - 4:00-7:00pm, 10 Evans
  - All material from last midterm and up to previous class
  - Includes virtual memory
  - One page of handwritten notes, both sides
- Final Exam
  - December $16^{th}$,8:00-11:00, Bechtel Auditorium
  - Covers whole course
  - Two pages of handwritten notes, both sides
- Final Topics: Any suggestions?



## Role of Testing



- Testing is basic to every engineering discipline
  - Design a drug
  - Manufacture an airplane
  - Etc.
- Why?
  - Because our ability to predict how our creations will behave is imperfect
  - We need to check our work, because we will make mistakes
- Some Testing Goals:
  - Reveal faults
  - Establish confidence
    - » of reliability
    - » of (probable) correctness
    - » of detection (therefore absence) of particular faults
  - Clarify/infer the specification

## Typical Software Licence

- 11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
- 12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.



## Independent Testing

- Programmers never believe they made mistake
  - Plus a vested interest in not finding mistakes
- Design and programming are constructive tasks
  - Testers must seek to break the software
- Wrong conclusions:
  - The developer should not be testing at all
    - » Instead: "Test before you code"
  - Testers get involved once software is done
    - » Instead: Testers involved at all stages
  - Toss the software over the wall for testing
    - » Instead: Testers and developers collaborate in developing the test suite
  - Testing team is responsible for assuring quality
    - » Instead: Quality is assured by a good software process



## Principles of Testability

- Testers have two jobs
  - Clarify the specification
  - Find (important) bugs
- Avoid unpredictable results
  - No unnecessary non-deterministic behavior
- Design in self-checking
  - Have system check its own work (Asserts!)
  - May require adding some redundancy to the code
- Avoid system state
  - System retains nothing across units of work
    - » A transaction, a session, etc.
  - System returns to well-known state after each task
    - » Easiest system to test (or to recover from failure)
- Minimize interactions between features
  - Number of interactions can easily grow huge
  - Rich breeding ground for bugs
- Have a test interface

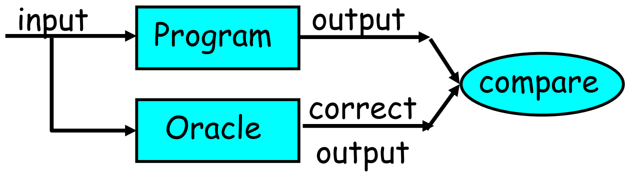


## Testing Frameworks

- Key components of a test system are
  - Building the system to test
    - » May build many different versions to test
  - Running the tests
  - Deciding whether tests passed/failed
    - » Sometimes a non-trivial task (e.g., compilers) !
  - Reporting results
- Testing frameworks provide these functions
  - E.g., Tinderbox, JUnit

## What is an Oracle?

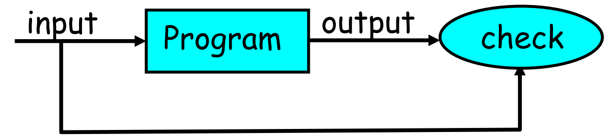
- Oracle = alternative realization of the specification




- Examples of oracles
  - The "eyeball oracle"
    - » Expensive, not dependable, lack of automation
  - A prototype, or sub-optimal implementation
    - » E.g., bubble-sort as oracle for quick sort
  - A manual list of expected results



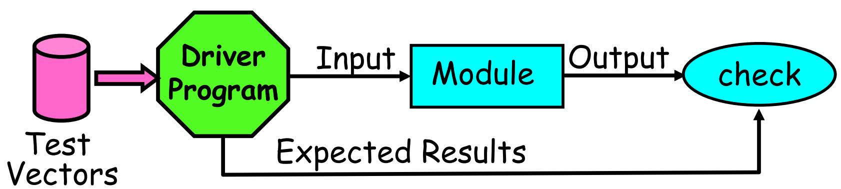
## Result Checking




- Easy to check the result of some algorithms
  - E.g., computing roots of polynomials, vs. checking that the result is correct
  - E.g., executing a query, vs. checking that the results meet the conditions
    - » Not easy to check that you got all results though !



## Data Driven Tests



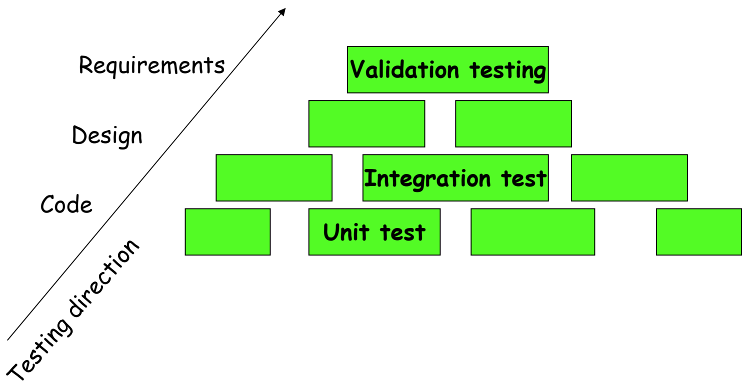

- Build a database of event tuples (or test vectors)
  - E.g: < Input1, Input2, Input3, Input4, Result>
  - So: <3, 4, "hello", 5, 42>
    <3, 5, "goodbye", 5, failure>
- A test is a series of such events chained together
  - Produce a high-level "driver" program to apply tuples to the system under test
    - » Tuples could be in a file and read in by driver program
  - Can be completely automatic



## Assertions

- Use assert(…) liberally
  - Documents important invariants
  - Makes your code self-checking
  - And does it on *every* execution !
  - You still have to worry about coverage
- May need to write functions that check invariants
- Opinion: Most programmers don't use assert enough

## Testing Strategies






## Unit Tests

- Focus on smallest unit of design
  - A procedure, a class, a component
- Test the following
  - Local data structures
  - Basic algorithm
  - Boundary conditions
  - Error handling
- Good idea to plan unit tests ahead



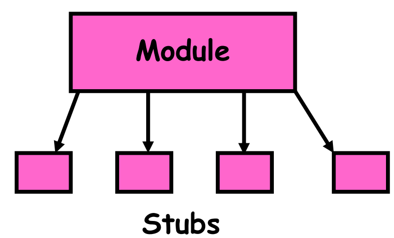
## Integration Testing

- If all parts work, how come the whole doesn't?
- For software, the whole is more than the sum of the parts
  - Individual imprecision is magnified (e.g., races)
  - Unclear interface design
- Don't try the "big bang" integration !
- Do incremental integration
  - Top-down integration
  - Bottom-up integration



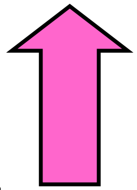
## Top-Down Integration

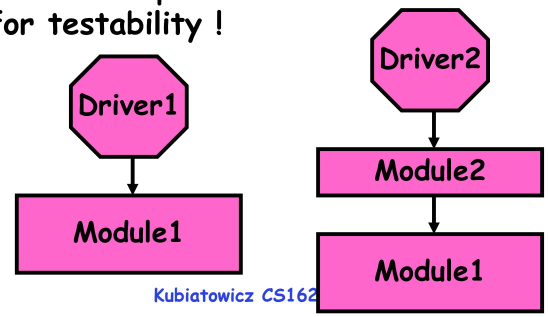
- Test the main control module first
- Slowly replace stubs with real code
  - Can go depth-first
    - » Along a favorite path, to create quickly a working system
  - Or, breadth first
- Problem: you may need complex stubs to test higher-levels



## Bottom-Up Integration

- Integrate already tested modules
- No stubs, but need drivers
  - Often the drivers are easier to write
- Example:
  - Financial code that depends on subroutine for computing roots of polynomials
  - We cannot test the code without the subroutine
    - » A simple stub might not be enough
  - We can develop and test the subroutine first
- Plan for testability !








## Stress Testing

- Push system into extreme situations
  - And see if it still works . . .
- Stress
  - Performance
    - » Feed data at very high rates
  - Interfaces
    - » Replace APIs with badly behaved stubs
  - Internal structures
    - » Works for any size array? Try sizes 0 and 1.
  - Resources
    - » Set memory artificially low.
    - » Same for # of file descriptors, network connections, etc.



## Stress Testing (Cont.)

- Stress testing will find many obscure bugs
  - Explores the corner cases of the design

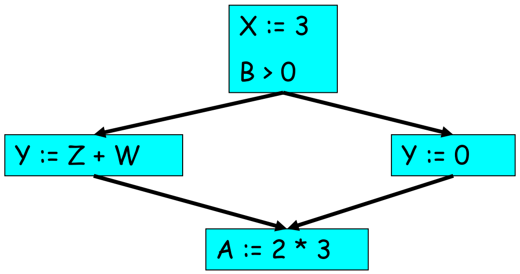
  "Bugs lurk in corners, and congregate at boundaries"
- Some may not be worth fixing
  - Bugs too unlikely to arise in practice
- A corner case now is tomorrow's common case
  - Data rates, data sizes always increasing
  - Your software will be stressed



## Code Coverage

- Code Coverage
  - Make sure that code is *covered*
- Control flow coverage criteria: Make sure you have tests that exercise every…
  - Statement (node, basic block) coverage
  - Branch (edge) and condition coverage
  - Data flow (syntactic dependency) coverage
- More sophisticated coverage criteria increase the #units to be covered in a program

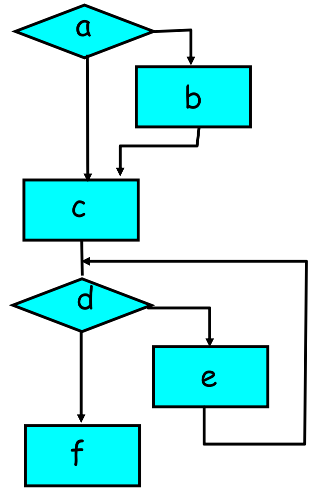
## Control Flow Graphs: The One Slide Tutorial




- *A graph*
- Nodes are ***basic blocks***
  - Maximal single-entry, jump-exit code segments
- Edges are transfers of control between basic blocks
  - E.g. Branches.



## Basic structural criteria (ex.)



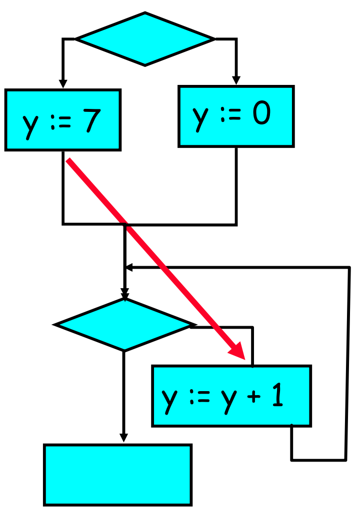

Edge ac is required by <u>all-edges</u> but not by <u>all-nodes</u> coverage

- abcdedf – all nodes

Typical <u>loop coverage</u> criterion would require zero iterations (cdf), one iteration (cdedf), and multiple iterations (cdededed...df)



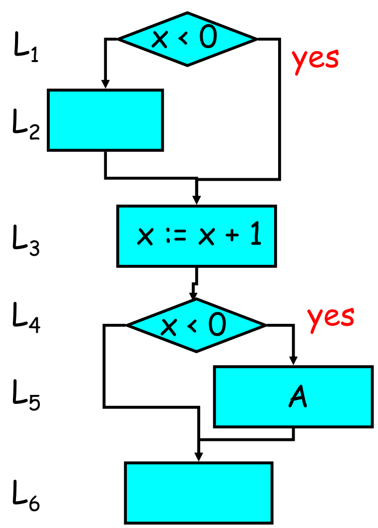
## Data flow coverage criteria (ex.)




- An untested def-use association could hide an erroneous computation
  - Even though we have all-node and all-edge coverage
- This suggests all paths coverage



## All Paths Coverage




- There could be an exponential number of paths in a acyclic program
  - 2 conditionals ⇒ 4 max combinations
- Many are not reachable:
  - $L_1$-$L_2$-$L_3$-$L_4$-$L_6$
- We choose
  - x = 0: $L_1$-$L_2$-$L_3$-$L_4$-$L_5$-$L_6$
  - x = -1: $L_1$-$L_3$-$L_4$-$L_6$
  - x = -2: $L_1$-$L_3$-$L_4$-$L_5$-$L_6$

## Code Coverage (Cont.)

- Code coverage has proven value
  - It's a real metric, though far from perfect
- But 100% coverage does not mean no bugs
  - Many bugs lurk in corner cases
  - E.g., a bug visible after loop executes 1,025 times
- And 100% coverage is almost never achieved
  - Products ship with < 60% coverage
  - High coverage may not even be economically desirable
    - » May be better to focus on tricky parts
- Code coverage helps identify weak test suites
  - Tricky bits with low coverage are a danger sign
  - Areas with low coverage suggest something is missing in the test suite



## Code Inspections

- Problem: Testing is weak
  - Can never test more than a tiny fraction of possibilities
  - Testers don't know as much about the code as the developers
    - » But developers can only do so much testing
- Here's an idea: Understand the code!
  - One person explains to a group of programmers how a piece of code works
- Key points
  - Don't try to read too much code at one sitting
    - » A few pages at most
  - Everyone comes prepared
    - » Distribute code beforehand
  - No blame
    - » Goal is to understand, clarify code, not roast programmers



## Experience with Inspections

- Inspections work!
  - Finds 70%-90% of bugs in studies
  - Dramatically reduces cost of finding bugs
- Other advantages
  - Teaches everyone the code
  - Finds bugs earlier than testing
- Bottom line: More than pays for itself

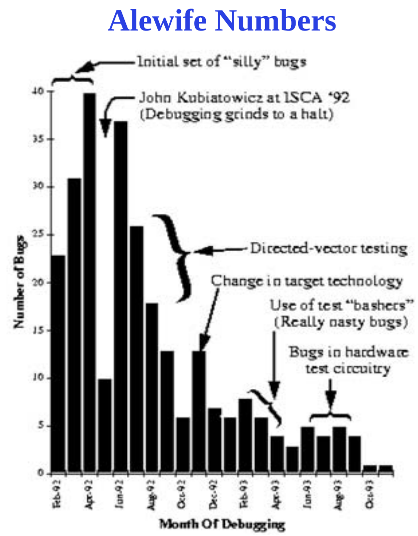


## Regression Testing

- Idea
  - When you find a bug,
  - Write a test that exhibits the bug,
  - And always run that test when the code changes,
  - So that the bug doesn't reappear
- Without regression testing, it is surprising how often old bugs reoccur
  - Regression testing ensures forward progress
  - We never go back to old bugs
- Regression testing can be manual or automatic
  - Ideally, run regressions after every change
  - To detect problems as quickly as possible
- But, regression testing is expensive
  - Limits how often it can be run in practice
  - Reducing cost is a long-standing research problem

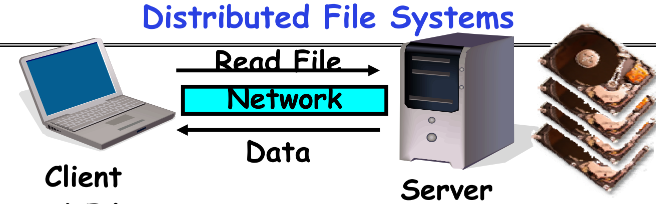
## Testing: When are you done?

- When you run out of time?
- Consider rate of bug finding
  - Rate is high ⇒ NOT DONE
  - Rate is low ⇒ May need new testing methodology
- Coverage Metrics
  - How well did you cover the design with test cases?
- Types of testing:
  - Directed Testing – test explicit behavior
  - Random Testing – apply random values or orderings
  - Daemons – continuous error/unexpected behavior insertion

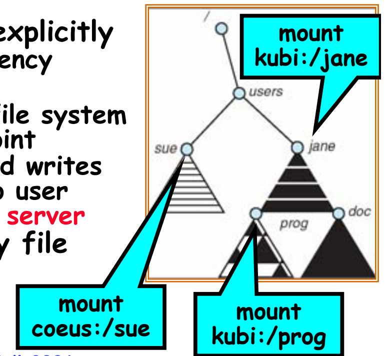





## Distributed File Systems

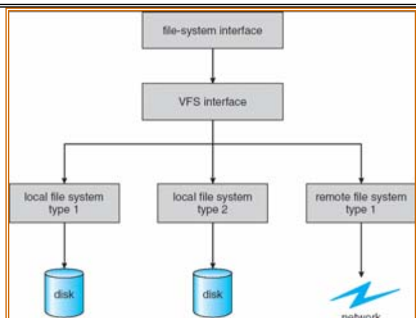



- Distributed File System:
  - Transparent access to files stored on a remote disk
- Naming choices (always an issue):
  - *Hostname*:localname: Name files explicitly
    - » No location or migration transparency
  - *Mounting* of remote file systems
    - » System manager mounts remote file system by 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





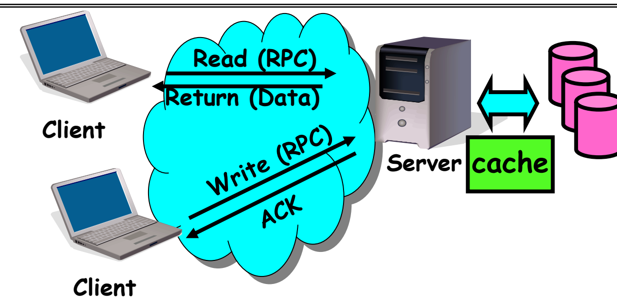

## Virtual File System (VFS)




- 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

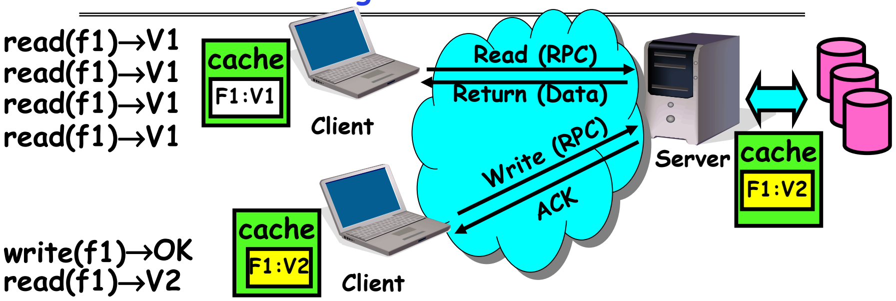


## Simple Distributed File System




- 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

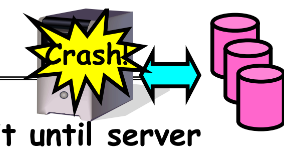
## Use of caching to reduce network load




- 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



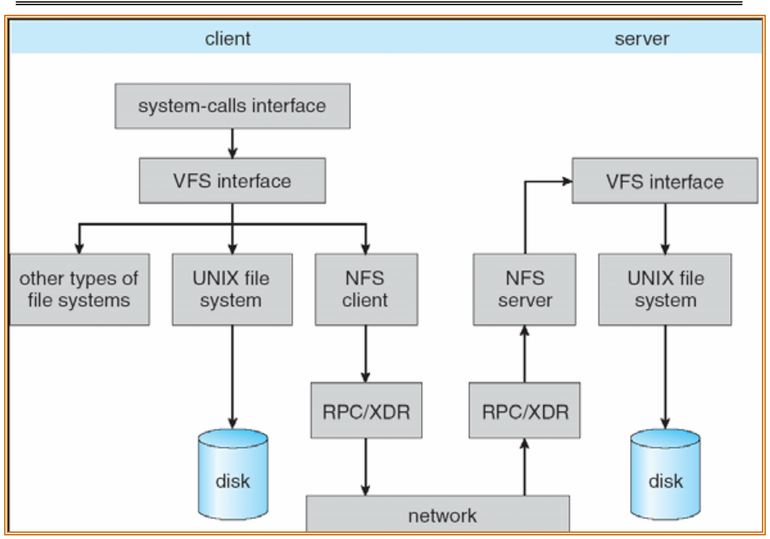
## Failures




- 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)

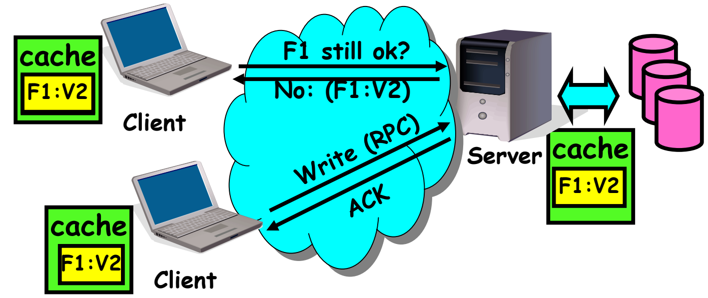
## NFS Continued

- 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.

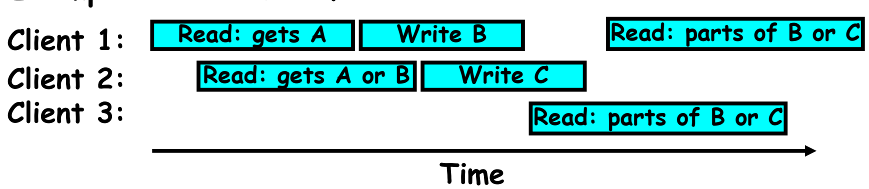



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!



## 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) → 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 ⇒ more files can be cached locally
  - Callbacks ⇒ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes→server, cache misses→server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost relative to workstation



## 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)
- Testing Goals
  - Reveal faults
  - Clarify Specification
- Testing Frameworks:
  - Provide mechanism for applying tests (driver), checking results, reporting problems
  - Oracle: simpler version of code for testing outputs
  - Assertions: Documents (and checks) important invariants
- Levels of Tests:
  - Unit testing: per module
  - Integration Testing: tying modules together
  - Regression Testing: making sure bugs don't reappear



## Conclusion (2)

- 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