CS162
Operating Systems and Systems Programming
Lecture 23

Distributed Systems

April 15, 2010
Benjamin Hindman
http://inst.eecs.berkeley.edu/~cs162

Distributed Systems are Everywhere!

- We need (want?) to share physical devices (e.g., printers) and information (e.g., files)
- Many applications are distributed in nature (e.g., ATM machines, airline reservations)
- Many large problems can be solved by decomposing into lots of smaller problems that can be run in parallel (e.g., MapReduce, SETI@home)

What makes building distributed systems interesting?

- Programming models
- Transparency
- Fault-tolerance
- Performance
- Scalability
- Consistency
- Security

Distributed Applications

- How do you actually program a distributed application?

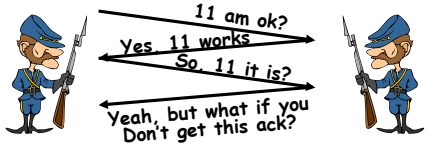
  - Use networking building blocks to provide a basic send/receive abstraction (message passing)
    - » Semantics: sender picks a specific receiver and receiver gets all or none of the message
    - » Queue incoming messages on receive side

Using Messages: Send/Receive behavior

- When should `send` return?
  - Asynchronous: return immediately
  - Synchronous: return after …
    - » Receiver gets message? (i.e., ack received)
    - » When message is safely buffered on destination?
    - » Right away, if message is buffered on source node?
- Main question here:
  - When can the sender be sure that receiver actually received the message?

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



Distributed Decision Making

- 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

Distributed Transactions

- Since we can't solve the General's Paradox, let's solve a related problem, distributed transaction: N machines agree to do something, or not do it, atomically
- Why should we care? Banks do this every day (every minute, every second, …)
- Two-Phase Commit Protocol
  - Phase 1, coordinator sends out a request to commit
    - » each participant responds with yes or no
  - Phase 2
    - » If everyone says yes, coordinator sends out a commit
    - » If someone says no, coordinator sends out an abort

Two-Phase Commit Details

- Each participant uses a local, persistent, corrupt-free log to keep track of whether a 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
  - Log can be used to complete this process such that all machines either commit or don't commit
  - Timeouts can be used to retry if coordinator doesn't hear from all participants

Two-Phase Commit Example

- Simple Example: A≡WellsFargo, B≡Chase
  - Phase 1:
    - » 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"

Two-Phase Commit Gotchas

- 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
- Alternatives such as "Three Phase Commit" don't have this blocking problem
- What happens if one or more of the participants is malicious?

Remote Procedure Call

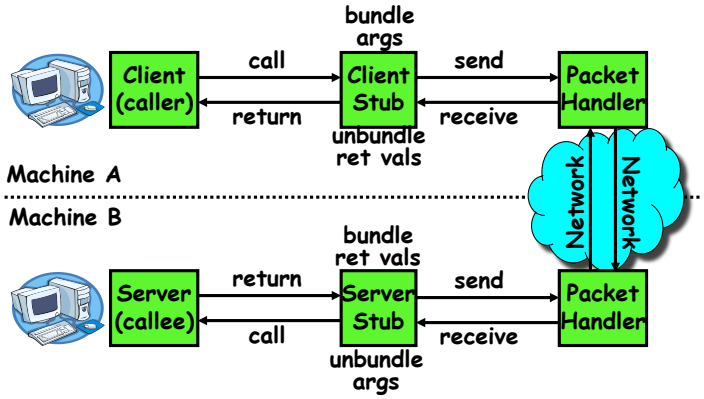
- Raw messaging is a bit too low-level for programming
- Another option: Remote Procedure Call (RPC)
  - Looks like a local procedure call on client:

    ```
    file.read(1024);
    ```
  - Translated automatically into a procedure call on remote machine (server)
- Implementation:
  - Uses request/response message passing "under the covers"

RPC Details

- Client and server use "stubs" to glue pieces together
  - 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.
  - Needs to account for cross-language and cross-platform issues
- Technique: compiler generated stubs
  - Input: interface definition language (IDL)
    - » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language

RPC Information Flow



RPC Binding

- How does client know which machine to send RPC?
  - Need to translate name of remote service into network endpoint (e.g., host:port)
  - 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
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one

RPC Transparency

- 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

Problems with RPC

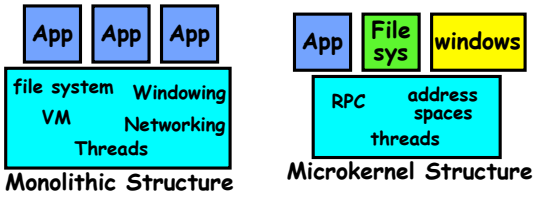
- Handling failures
  - Different failure modes in distributed system than on a single machine
  - Without RPC a failure within a procedure call usually meant whole application would crash/die
  - With RPC a failure within a procedure call means remote machine crashed, but local one could continue working
  - Answer? Distributed transactions can help
- Performance
  - Cost of Procedure call « same-machine RPC « network RPC
  - Means programmers must be aware they are using RPC (so much for transparency!)
    - » Caching can help, but may make failure handling even more complex

Administrivia

- Should be working on Project 4
  - Last one!
- Do Project 3 Group Evaluations ASAP

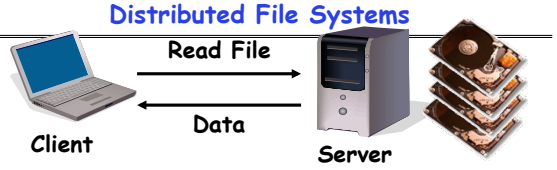
Microkernel operating systems

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



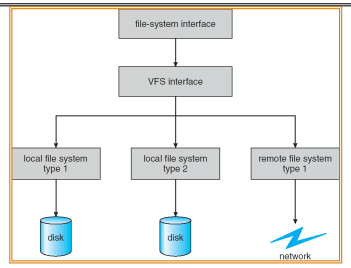
- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - » For example in the X windowing system: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

Distributed File Systems



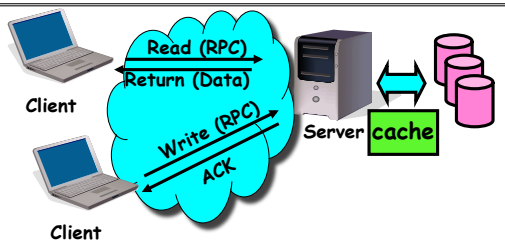
- Distributed File System:
  - Transparent access to files stored on a remote disk
- What's the basic abstraction?
  - Keep reads and writes look the same, even though they operate on remote files (transparency)
- Naming
  - How should the files be named?
  - Do those names imply a location?

Virtual File System (VFS)



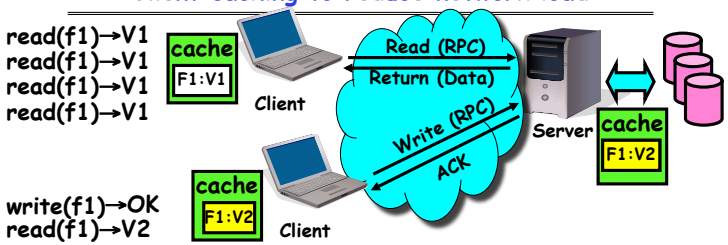
- VFS: Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
- VFS allows the same system call interface to be used for different types of file systems (local AND remote)

Simple Distributed File System



- EVERY read and write gets forwarded to server
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Server can be a bottleneck

Client caching to reduce network load



- 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

Network File System (NFS)

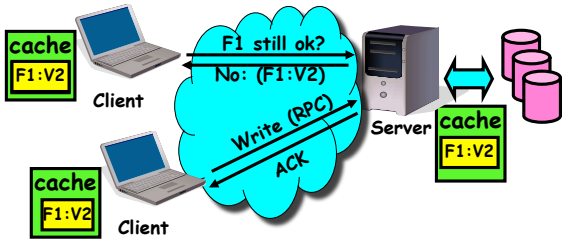
- Three Layers for NFS system
  - Use open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
- 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/stay consistent!

NFS Continued

- NFS servers are stateless; each request provides all arguments required for execution
  - No need to perform network open() or close() on file – each operation stands on its own
  - If server crashes, client can retry operation when server comes back up!
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Can just re-read or re-write file block – no side effects
  - What about "remove"? NFS does operation twice and second time returns an advisory error
- Failure Model:
  - Hang until server comes back up (next week?)
  - Return an error (oops, so much for transparency … 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



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

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()
  - Thus, 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?"
- For both AFS and NFS: central server is bottleneck
  - Relative to NFS, AFS has less server load:
    - » Disk as cache ⇒ more files can be cached locally
    - » Callbacks ⇒ server not involved if file is read-only
  - Regardless, all writes→server, cache misses→server
  - Server is single point of failure!

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

- Two-phase commit: distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit
- 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

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);
```

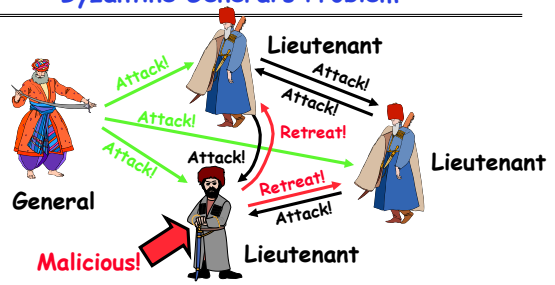
Request File

Get Response

Receive Request

Send Response

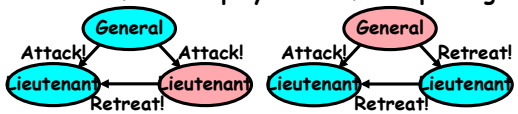
Byzantine General's Problem



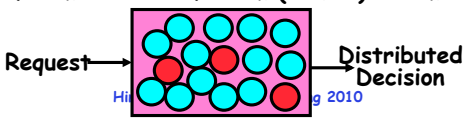
- Byzantine General's Problem (n players):
  - One General
  - n-1 Lieutenants
  - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is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

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



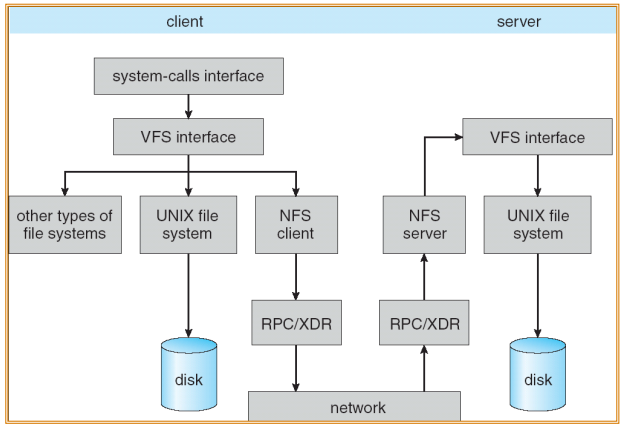
  - With f faults, need n > 3f to solve problem
- Various algorithms exist to solve problem
  - Original algorithm has #messages exponential in n
  - Newer algorithms have message complexity $O(n^2)$
    - » One from MIT, for instance (Castro and Liskov, 1999)
- Use of BFT (Byzantine Fault Tolerance) algorithm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



Dealing with Failures

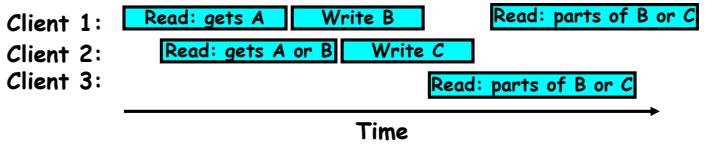
- What if server crashes? Can client wait until server comes back up and continue as before?
  - Any data in server memory but not on disk can be lost
  - Shared state across RPC: What if server crashes after seek? Then, when client does "read", it will fail
  - Message retries: suppose server crashes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



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