CS162
Operating Systems and Systems Programming
Lecture 23

Networking (Con't), Distributed File Systems, Key-Value stores

April 23rd, 2020

Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

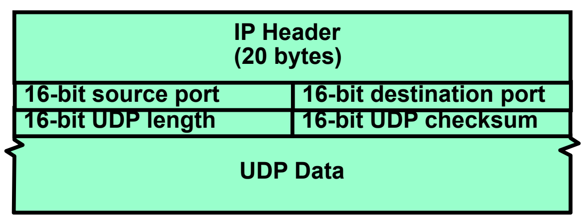
Recall: Network Layering

- Layering: building complex services from simpler ones
  - Each layer provides services needed by higher layers by utilizing services provided by lower layers
- The physical/link layer is pretty limited
  - Packets are of limited size (called the "Maximum Transfer Unit or MTU: often 200-1500 bytes in size)
  - Routing is limited to within a physical link (wire) or perhaps through a switch
- Our goal in the following is to show how to construct a secure, ordered, message service routed to anywhere:

Physical Reality: Packets	Abstraction: Messages
Limited Size	Arbitrary Size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-machine	Process-to-process
Only on local area net	Routed anywhere
Asynchronous	Synchronous
Insecure	Secure

Recall: UDP Transport Protocol

- The Unreliable Datagram Protocol (UDP)
  - Layered on top of basic IP (IP Protocol 17)
  - Datagram: an unreliable, unordered, packet sent from source user → dest user (Call it UDP/IP)

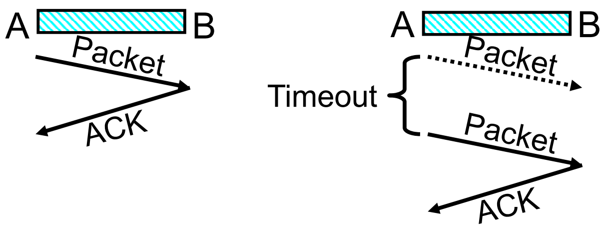


  - UDP adds minimal header to deliver from process to process (i.e. the source and destination Ports)
- Important aspect: low overhead!
  - Often used for high-bandwidth video streams
  - Many uses of UDP considered "anti-social" – none of the "well-behaved" aspects of (say) TCP/IP

Reliable Message Delivery: the Problem

- All physical networks can garble and/or drop packets
  - Physical media: packet not transmitted/received
    - » If transmit close to maximum rate, get more throughput – even if some packets get lost
    - » If transmit at lowest voltage such that error correction just starts correcting errors, get best power/bit
  - Congestion: no place to put incoming packet
    - » Point-to-point network: insufficient queue at switch/router
    - » Broadcast link: two host try to use same link
    - » In any network: insufficient buffer space at destination
    - » Rate mismatch: what if sender send faster than receiver can process?
- Reliable Message Delivery on top of Unreliable Packets
  - Need some way to make sure that packets actually make it to receiver
    - » Every packet received at least once
    - » Every packet received at most once
  - Can combine with ordering: every packet received by process at destination exactly once and in order

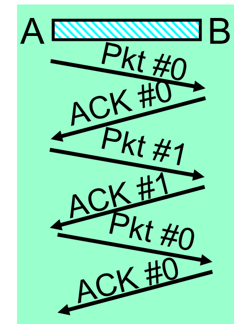
Using Acknowledgements



- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending "ACK") when packet received properly at destination
  - Timeout at sender: if no ACK, retransmit
- Some questions:
  - If the sender doesn't get an ACK, does that mean the receiver didn't get the original message?
    - » No
  - What if ACK gets dropped? Or if message gets delayed?
    - » Sender doesn't get ACK, retransmits, Receiver gets message twice, ACK each

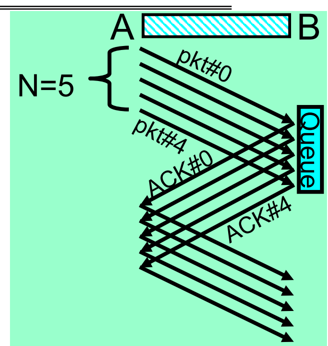
How to Deal with Message Duplication?

- Solution: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate number's; Discard if detected
- Requirements:
  - Sender keeps copy of unACK'd messages
    - » Easy: only need to buffer messages
  - Receiver tracks possible duplicate messages
    - » Hard: when ok to forget about received message?
- Alternating-bit protocol:
  - Send one message at a time; don't send next message until ACK received
  - Sender keeps last message; receiver tracks sequence number of last message received
- Pros: simple, small overhead
- Con: Poor performance
  - Wire can hold multiple messages; want to fill up at (wire latency × throughput)
- Con: doesn't work if network can delay or duplicate messages arbitrarily

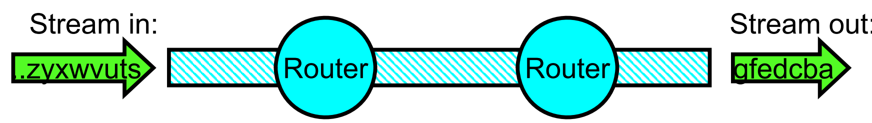


Better Messaging: Window-based Acknowledgements

- Windowing protocol (not quite TCP):
  - Send up to N packets without ack
    - » Allows pipelining of packets
    - » Window size (N) < queue at destination
  - Each packet has sequence number
    - » Receiver acknowledges each packet
    - » ACK says "received all packets up to sequence number X"/send more
- ACKs serve dual purpose:
  - Reliability: Confirming packet received
  - Ordering: Packets can be reordered at destination
- What if packet gets garbled/dropped?
  - Sender will timeout waiting for ACK packet
    - » Resend missing packets ⇒ Receiver gets packets out of order!
  - Should receiver discard packets that arrive out of order?
    - » Simple, but poor performance
  - Alternative: Keep copy until sender fills in missing pieces?
    - » Reduces # of retransmits, but more complex
- What if ACK gets garbled/dropped?
  - Timeout and resend just the un-acknowledged packets

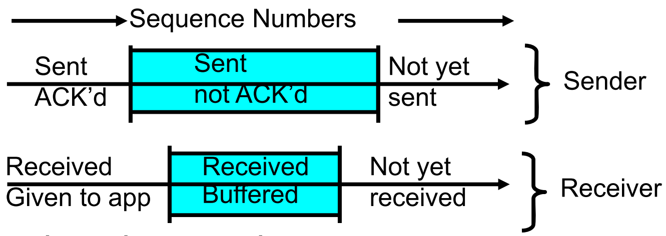


Transmission Control Protocol (TCP)



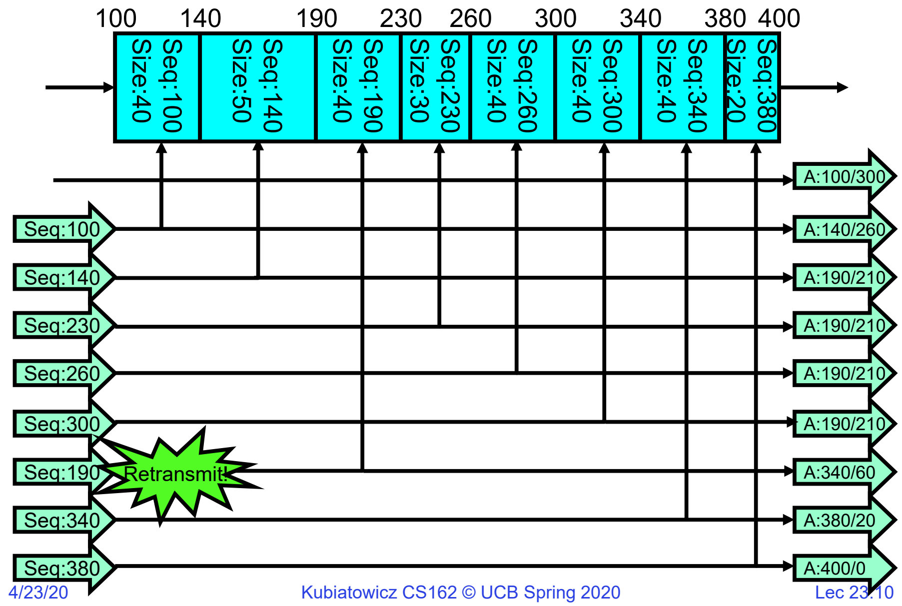
- Transmission Control Protocol (TCP)
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- TCP Details
  - Fragments byte stream into packets, hands packets to IP
    - » IP may also fragment by itself
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

TCP Windows and Sequence Numbers



- Sender has three regions:
  - Sequence regions
    - » sent and ACK'd
    - » sent and not ACK'd
    - » not yet sent
  - Window (colored region) adjusted by sender
- Receiver has three regions:
  - Sequence regions
    - » received and ACK'd (given to application)
    - » received and buffered
    - » not yet received (or discarded because out of order)

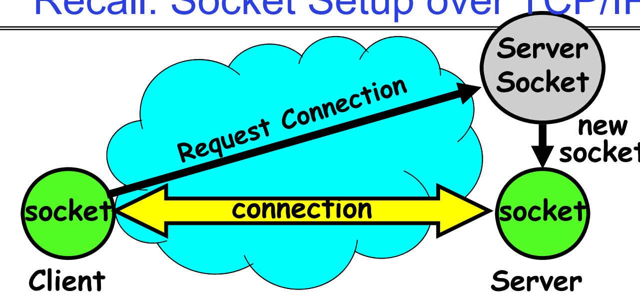
Window-Based Acknowledgements (TCP)



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*"

Recall: Socket Setup over TCP/IP



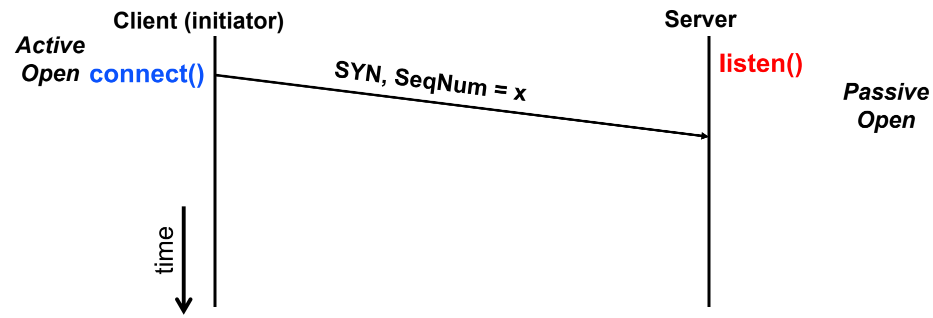
- Things to remember:
  - Connection involves 5 values:
    [Client Addr, Client Port, Server Addr, Server Port, Protocol]
  - Often, Client Port "randomly" assigned
  - Server Port often "well known"
    - » 80 (web), 443 (secure web), 25 (sendmail), etc
    - » Well-known ports from 0—1023
- Network Address Translation (NAT) allows many internal connections (and/or hosts) with a single external IP address

Open Connection: 3-Way Handshaking

- Goal: agree on a set of parameters, i.e., the start sequence number for each side
  - Starting sequence number (first byte in stream)
  - Must be unique!
    - » 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

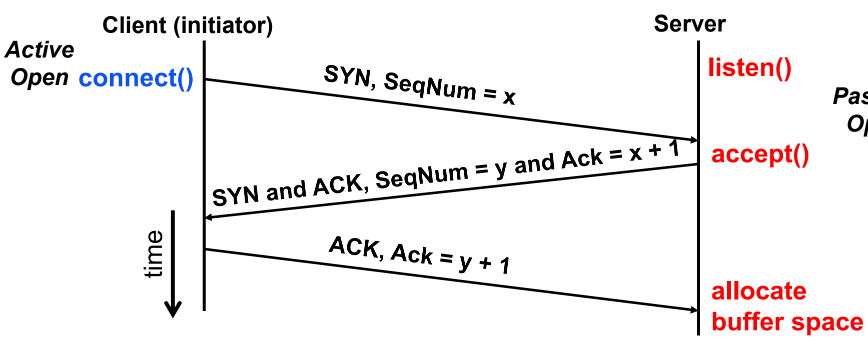
Open Connection: 3-Way Handshaking

- Server waits for new connection calling listen()
- Sender call connect() passing socket which contains server's IP address and port number
  - OS sends a special packet (SYN) containing a proposal for first sequence number, x

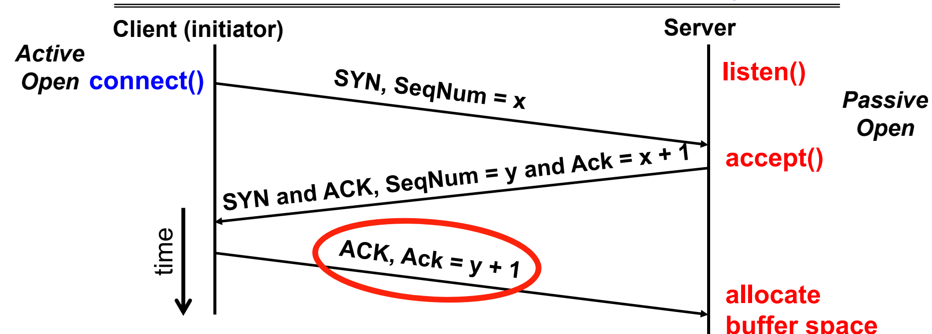


Open Connection: 3-Way Handshaking

- If it has enough resources, server calls accept() to accept connection, and sends back a SYN ACK packet containing
  - Client's sequence number incremented by one, (x + 1)
    - » Why is this needed?
  - A sequence number proposal, y, for first byte server will send



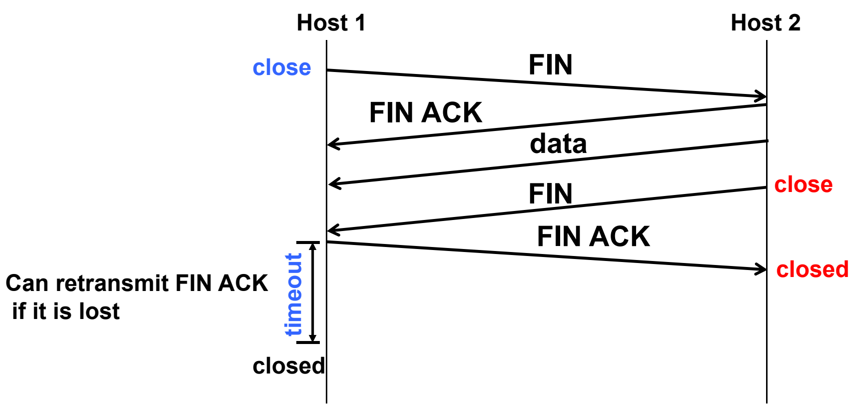
Denial of Service Vulnerability



- SYN attack: send a huge number of SYN messages
  - Causes victim to commit resources (768 byte TCP/IP data structure)
- Alternatives: Do not commit resources until receive final ACK
  - *SYN Cache*: when SYN received, put small entry into cache (using hash) and send SYN/ACK, If receive ACK, then put into listening socket
  - *SYN Cookie*: when SYN received, encode connection info into sequence number/other TCP header blocks, decode on ACK

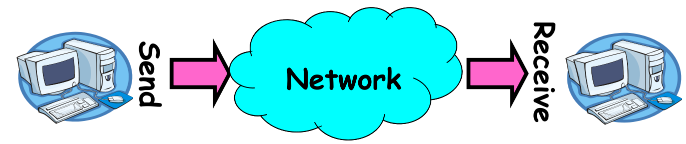
Close Connection

- Goal: both sides agree to close the connection
- 4-way connection tear down



Recall: Distributed System Protocols are Built with Message Passing

- How do you actually program a distributed application?
  - Multiple threads, running on different machines
    - » How do they coordinate and communicate

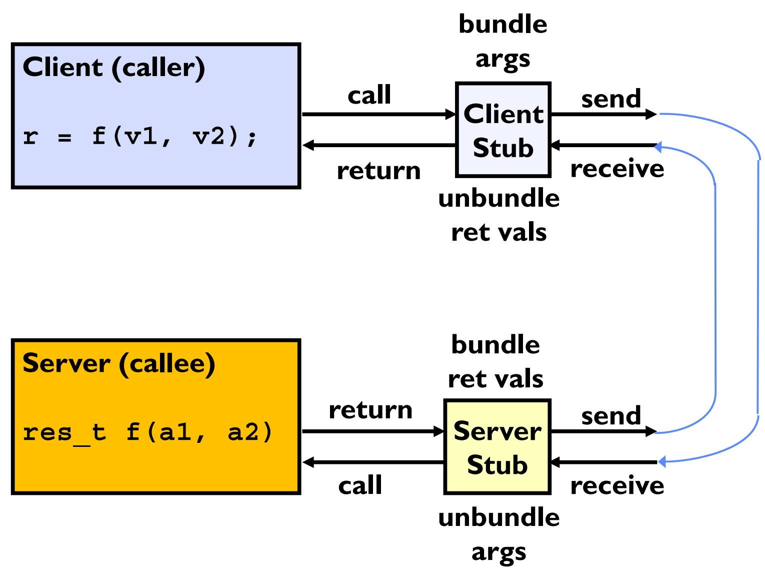


  - send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox: temporary holding area for messages
    - » Includes both destination location and queue
  - `Send(message,mbox)`
    - » Send message to remote mailbox 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

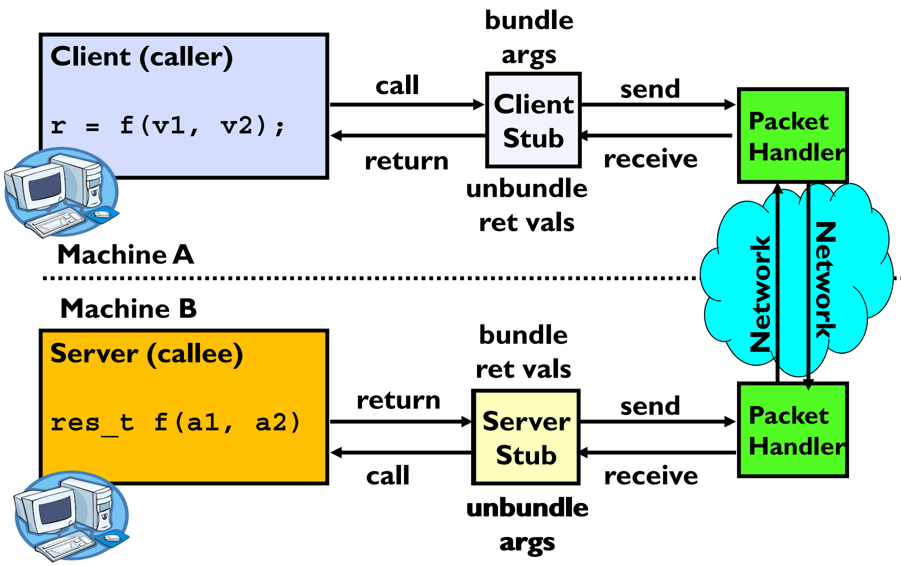
Remote Procedure Call (RPC)

- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
  - And – what about machines with different byte order ("BigEndian" vs "LittleEndian")
- Another 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");

RPC Concept



RPC Information Flow



RPC Implementation

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

RPC Details (1/3)

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)
- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    - » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - » Code for server to unpack message, call procedure, pack results, send them off

RPC Details (2/3)

- Cross-platform issues:
  - What if client/server machines are different architectures/ languages?
    - » Convert everything to/from some canonical form
    - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions)
- How does client know which mbox (destination queue) to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    - » This is another word for "naming" at network level
    - » Static: fixed at compile time
    - » Dynamic: performed at runtime

RPC Details (3/3)

- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynamic translation of service → mbox
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one
- What if there are multiple servers?
  - Could give flexibility at binding time
    - » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - » Choose unloaded server for each new request
    - » Only works if no state carried from one call to next
- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC: Non-Atomic Failures

- Different failure modes in dist. system than on a single machine
- Consider many different types of failures
  - User-level bug causes address space to crash
  - Machine failure, kernel bug causes all processes on same machine to fail
  - Some machine is compromised by malicious party
- Before RPC: whole system would crash/die
- After RPC: One machine crashes/compromised while others keep working
- Can easily result in inconsistent view of the world
  - Did my cached data get written back or not?
  - Did server do what I requested or not?
- Answer? Distributed transactions/Byzantine Commit

Problems with RPC: Performance

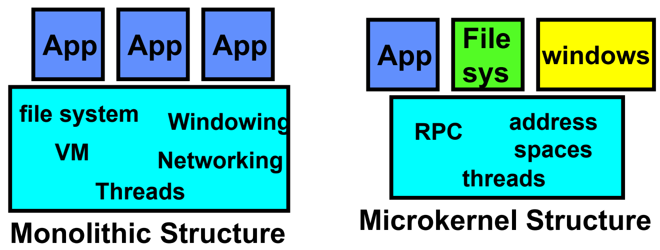
- RPC is *not* performance transparent:
  - Cost of Procedure call « same-machine RPC « network RPC
  - Overheads: Marshalling, Stubs, Kernel-Crossing, Communication
- Programmers must be aware that RPC is not free
  - Caching can help, but may make failure handling complex

Cross-Domain Communication/ Location Transparency

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

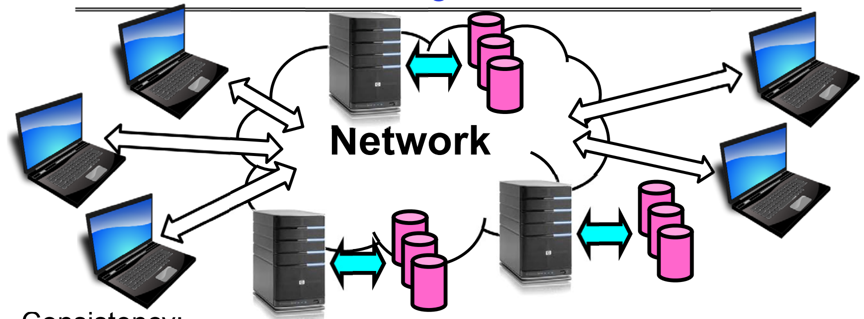
Microkernel operating systems

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



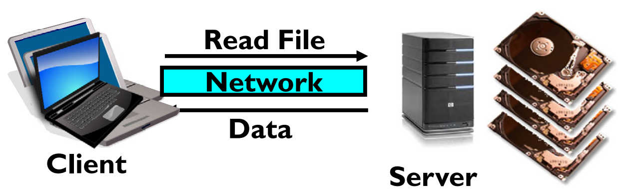
- Why split the OS into separate domains?
  – Fault isolation: bugs are more isolated (build a firewall)
  – Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  – Location transparent: service can be local or remote
    » For example in the X windowing system: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

Network-Attached Storage and the CAP Theorem



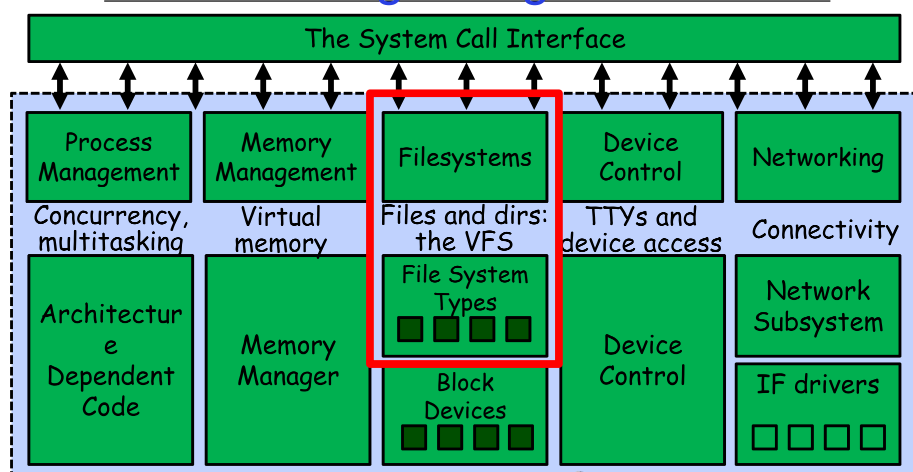
- Consistency:
  – Changes appear to everyone in the same serial order
- Availability:
  – Can get a result at any time
- Partition-Tolerance
  – System continues to work even when network becomes partitioned
- Consistency, Availability, Partition-Tolerance (CAP) Theorem: Cannot have all three at same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

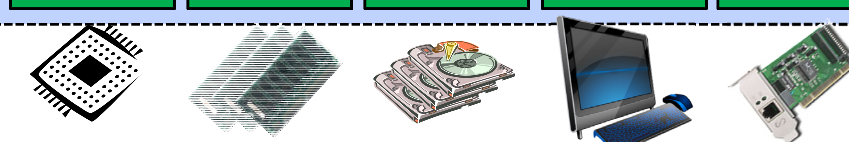
Distributed File Systems



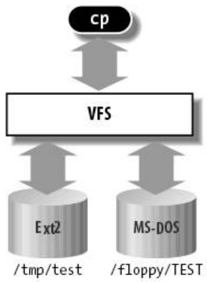
- Transparent access to files stored on a remote disk
- *Mount* remote files into your local file system
  – Directory in local file system refers to remote files
  – e.g., `/home/oksi/162/` on laptop actually refers to `/users/oski` on campus file server

Enabling Design: VFS





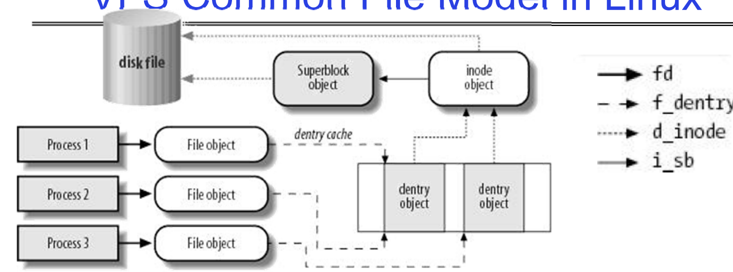
Virtual Filesystem Switch (Con't)




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
        O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

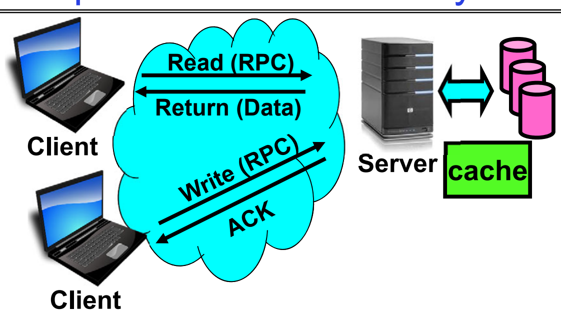
- VFS: Virtual abstraction similar to local file system
  - Provides virtual superblocks, inodes, files, etc
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used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

VFS Common File Model in Linux



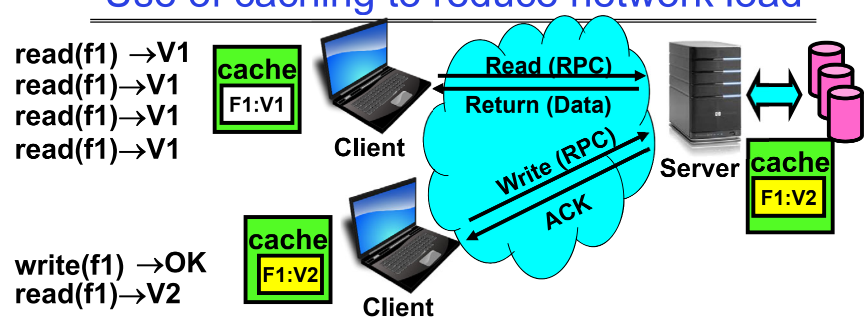
- Four primary object types for VFS:
  - superblock object: represents a specific mounted filesystem
  - inode object: represents a specific file
  - dentry object: represents a directory entry
  - file object: represents open file associated with process
- There is no specific directory object (VFS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

Simple Distributed File System



- Remote Disk: Reads and writes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
  - No local caching/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

Dealing with Failures

- What if server crashes? Can client wait until it comes back and just continue making requests?
  - Changes in server's cache but not in disk are lost
- What if there is shared state across RPC's?
  - Client opens file, then does a seek
  - Server crashes
  - What if client wants to do another read?
- Similar problem: What if client removes a file but server crashes before acknowledgement?

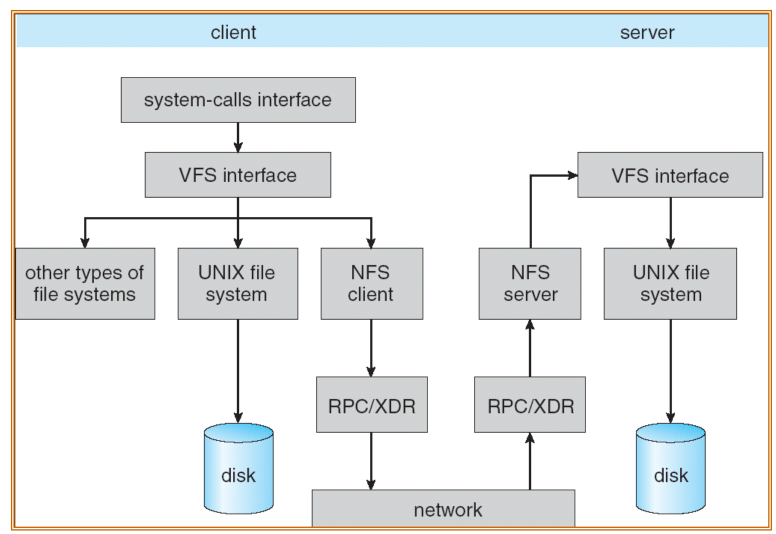
Stateless Protocol

- A protocol in which all information required to service a request is included with the request
- Even better: Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)
- Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)
- Recall HTTP: Also a stateless protocol
  - Include cookies with request to simulate a session

Network File System (Sun)

- Defines an RPC protocol for clients to interact with a file server
  - E.g., read/write files, traverse directories, …
  - Stateless to simplify failure cases
- Keeps most operations idempotent
  - Even removing a file: Return advisory error second time
- Don't buffer writes on server side cache
  - Reply with acknowledgement only when modifications reflected on disk

NFS Architecture



Network File System (NFS)

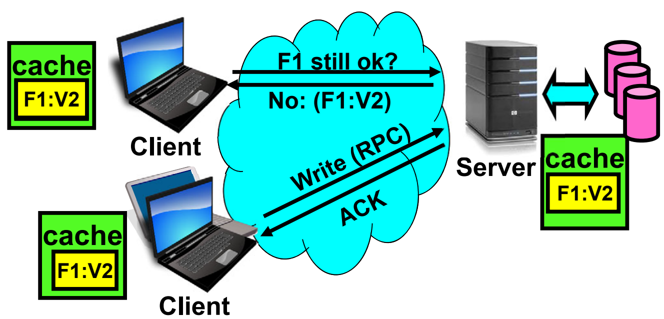
- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements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

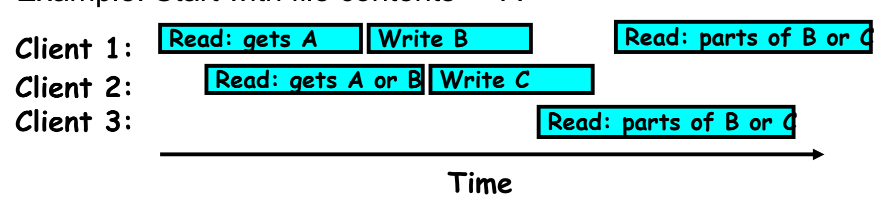
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version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

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

Sharing Data, rather than Files ?

- Key:Value stores are used everywhere
- Native in many programming languages
  - Associative Arrays in Perl
  - Dictionaries in Python
  - Maps in Go
  - …
- What about a collaborative key-value store rather than message passing or file sharing?
- Can we make it scalable and reliable?

Key Value Storage

Simple interface

- **`put(key, value);`** `// Insert/write "value" associated with key`
- **`get(key);`** `// Retrieve/read value associated with key`

Why Key Value Storage?

- Easy to Scale
  - Handle huge volumes of data (e.g., petabytes)
  - Uniform items: distribute easily and roughly equally across many machines
- Simple consistency properties
- Used as a simpler but more scalable "database"
  - Or as a building block for a more capable DB

Key Values: Examples

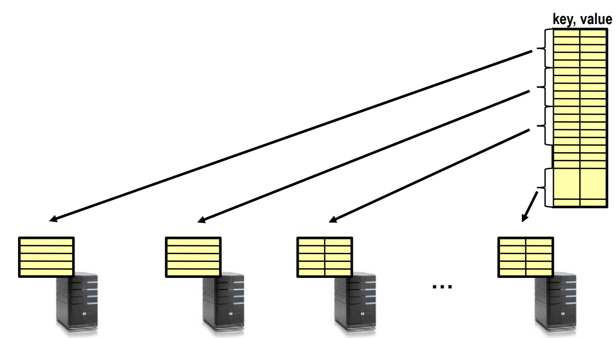
- Amazon:
  - Key: customerID
  - Value: customer p history, credit card, ..)
- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)
- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

Key-value storage systems in real life

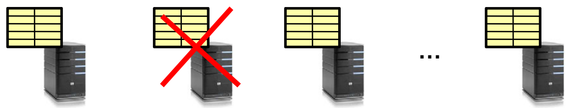
- **Amazon**
  - DynamoDB: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)
- **BigTable/HBase/Hypertable:** distributed, scalable data storage
- **Cassandra**: “distributed data management system” (developed by Facebook)
- **Memcached:** in-memory key-value store for small chunks of arbitrary data (strings, objects)
- **eDonkey/eMule:** peer-to-peer sharing system
- …

Key Value Store

- Also called Distributed Hash Tables (DHT)
- Main idea: partition set of key-values across many machines



Challenges



- **Scalability:**
  – Need to scale to thousands of machines
  – Need to allow easy addition of new machines
- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance
- **Consistency:** maintain data consistency in face of node failures and message losses
- **Heterogeneity** (if deployed as peer-to-peer systems):
  – Latency: 1ms to 1000ms
  – Bandwidth: 32Kb/s to 100Mb/s

Important Questions

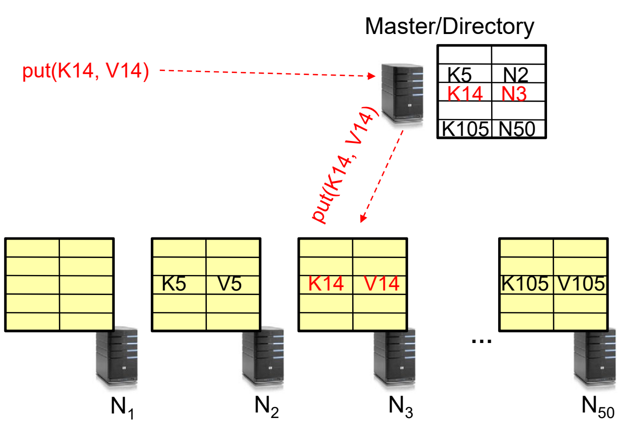
- **put(key, value)**:
  – where do you store a new (key, value) tuple?
- **get(key)**:
  – where is the value associated with a given "key" stored?
- And, do the above while providing
  – Scalability
  – Fault Tolerance
  – Consistency

How to solve the "where?"

- Hashing
  – But what if you don't know who are all the nodes that are participating?
  – Perhaps they come and go …
  – What if some keys are really popular?
  – More extended discussion – a bit later.
- Lookup
  – Hmm, won't this be a bottleneck and single point of failure?

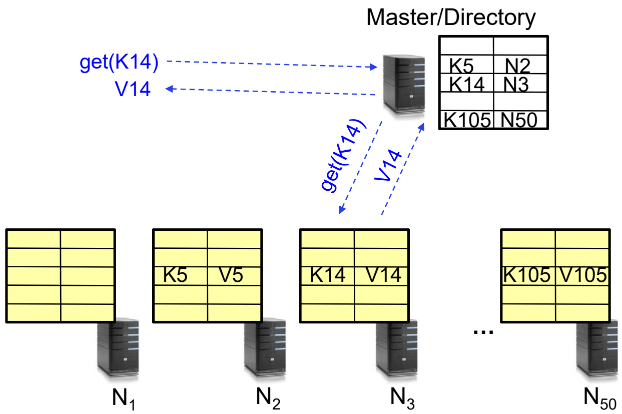
Recursive Directory Architecture (put)

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



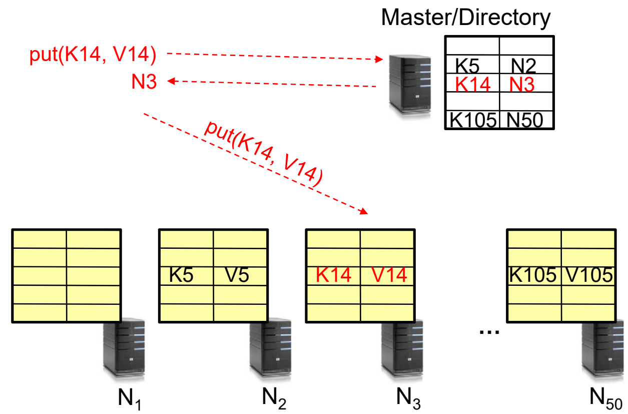
Recursive Directory Architecture (get)

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



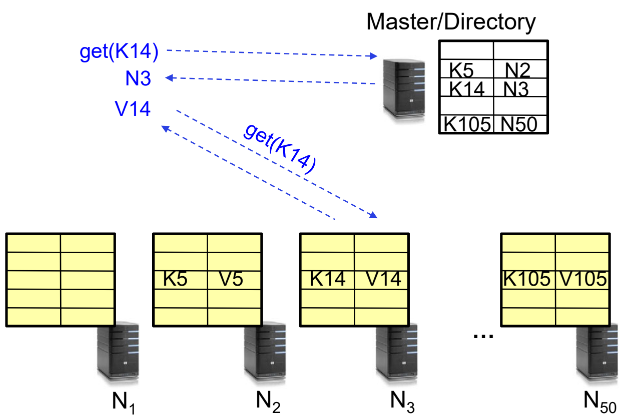
Iterative Directory Architecture (put)

- Having the master relay the requests → **recursive query**
- Another method: **iterative query** (this slide)
  - Return node to requester and let requester contact node

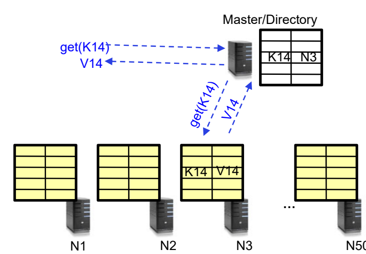


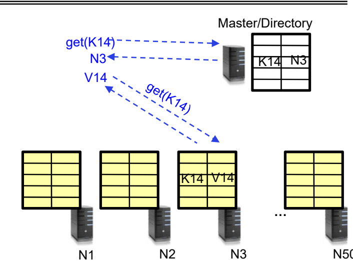
Iterative Directory Architecture (get)

- Having the master relay the requests → **recursive query**
- Another method: **iterative query** (this slide)
  - Return node to requester and let requester contact node



Iterative vs. Recursive Query





Recursive

+ Faster, as directory server is typically close to storage nodes
+ Easier for consistency: directory can enforce an order for all puts and gets
- Directory is a performance bottleneck

Iterative

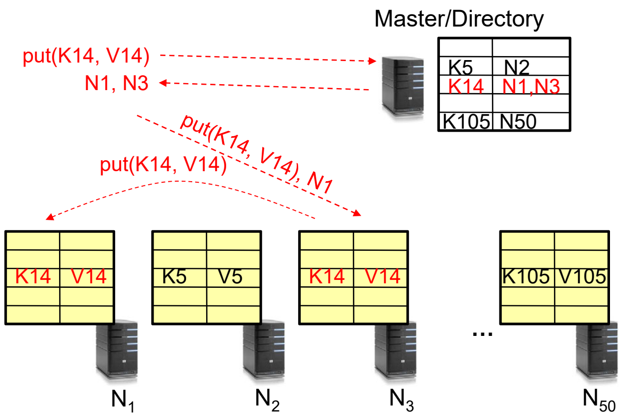
+ More scalable, clients do more work
- Harder to enforce consistency

Scalability: Is it easy to make the system bigger?

- Storage: Use more nodes
- Number of Requests
  - Can serve requests from all nodes on which a value is stored in parallel
  - Master can replicate a popular item on more nodes
- Master/Directory Scalability
  - Replicate It (multiple identical copies)
  - Partition it, so different keys are served by different directories
    - » But how do we do this….?

Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures

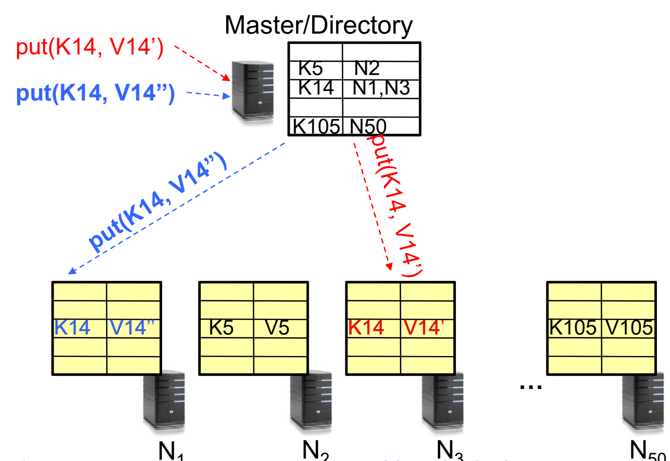


Consistency

- Need to make sure that a value is replicated correctly
- How do you know a value has been replicated on every node?
  - Wait for acknowledgements from every node
- What happens if a node fails during replication?
  - Pick another node and try again
- What happens if a node is slow?
  - Slow down the entire put()? Pick another node?
- In general, with multiple replicas
  - Slow puts and fast gets

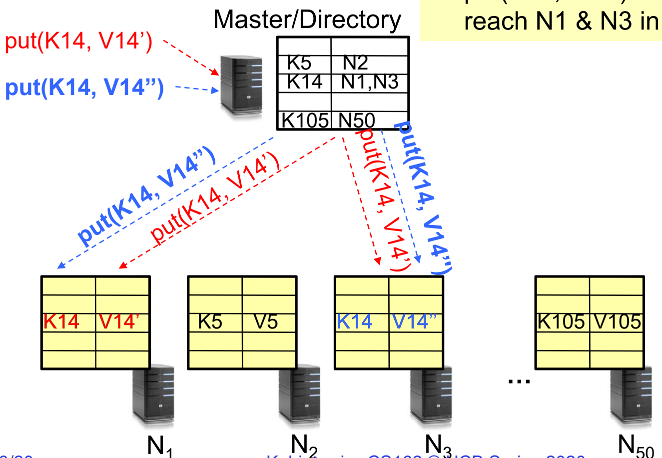
Consistency (cont'd)

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



Consistency (cont'd)

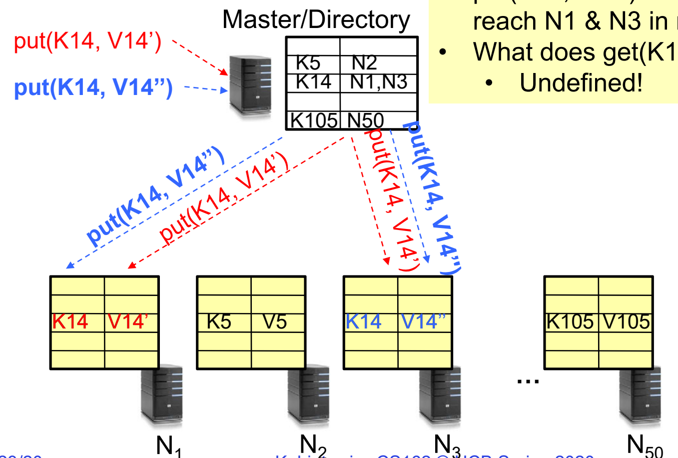
- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14'') reach N1 & N3 in reverse order!

Consistency (cont'd)

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14'') reach N1 & N3 in reverse order!
- What does get(K14) return?
  - Undefined!

Large Variety of Consistency Models

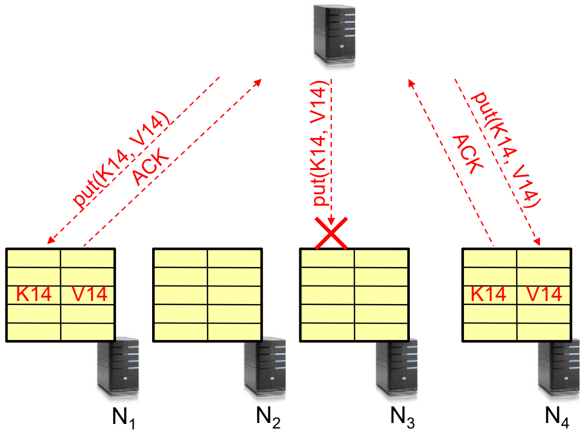
- Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
  – Think "one updated at a time"
  – Transactions
- Eventual consistency: given enough time all updates will propagate through the system
  – One of the weakest form of consistency; used by many systems in practice
  – Must eventually converge on single value/key (coherence)
- *And many others: causal consistency, sequential consistency, strong consistency, …*

Quorum Consensus

- Improve put() and get() operation performance
- Define a replica set of size N
  – put() waits for acknowledgements from at least W replicas
  – get() waits for responses from at least R replicas
  – W+R > N
- Why does it work?
  – There is at least one node that contains the update
- Why might you use W+R > N+1?

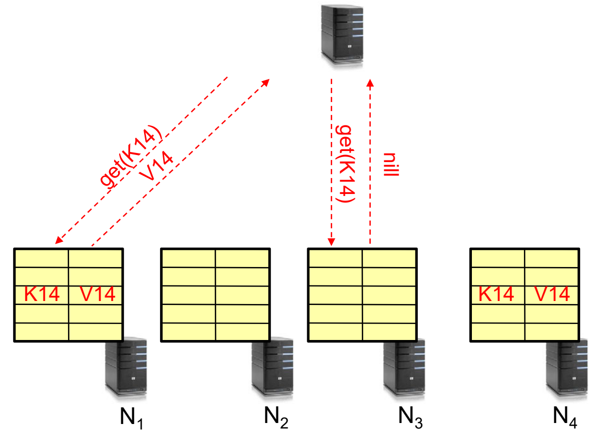
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N2, N4}
- Assume put() on N3 fails



Quorum Consensus Example

- Now, issuing get() to any two nodes out of three will return the answer



Scalability

- Storage: use more nodes
- Number of requests:
  - Can serve requests from all nodes on which a value is stored in parallel
  - Master can replicate a popular value on more nodes
- Master/directory scalability:
  - Replicate it
  - Partition it, so different keys are served by different masters/directories
    - » How do you partition?

Scalability: Load Balancing

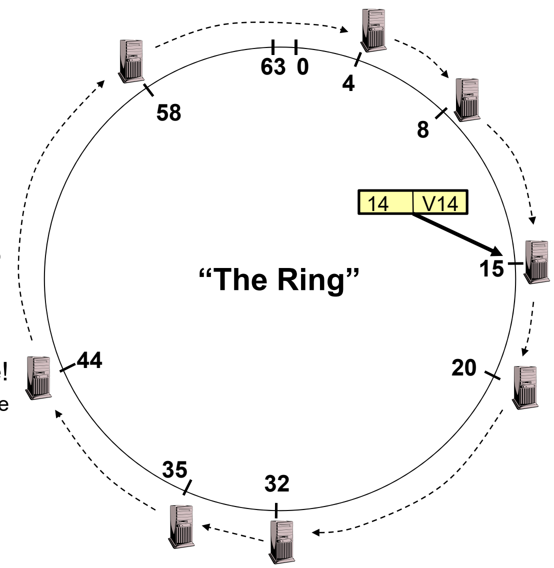
- Directory keeps track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available
- What happens when a new node is added?
  - Cannot insert only new values on new node. Why?
  - Move values from the heavy loaded nodes to the new node
- What happens when a node fails?
  - Need to replicate values from fail node to other nodes

Scaling Up Directory

- Challenge:
  – Directory contains a number of entries equal to number of (key, value) tuples in the system
  – Can be tens or hundreds of billions of entries in the system!
- Solution: **Consistent Hashing**
  – **Provides mechanism to divide [key,value] pairs amongst a (potentially large!) set of machines (nodes) on network**
- Associate to each node a unique *id* in an *uni*-dimensional space $0..2^m-1 \Rightarrow$ Wraps around: Call this "the ring!"
  – Partition this space across *n* machines
  – Assume keys are in same uni-dimensional space
  – Each [Key, Value] is stored at the node with the smallest ID larger than Key

Key to Node Mapping Example

- Paritioning example with m = 6 → ID space: 0..63
  – Node 8 maps keys [5,8]
  – Node 15 maps keys [9,15]
  – Node 20 maps keys [16, 20]
  – …
  – Node 4 maps keys [59, 4]
- For this example, the mapping [14, V14] maps to node with ID=15
  – Node with smallest ID larger than 14 (the key)
- In practice, m=256 or more!
  – Uses cryptographically secure hash such as SHA-256 to generate the node IDs

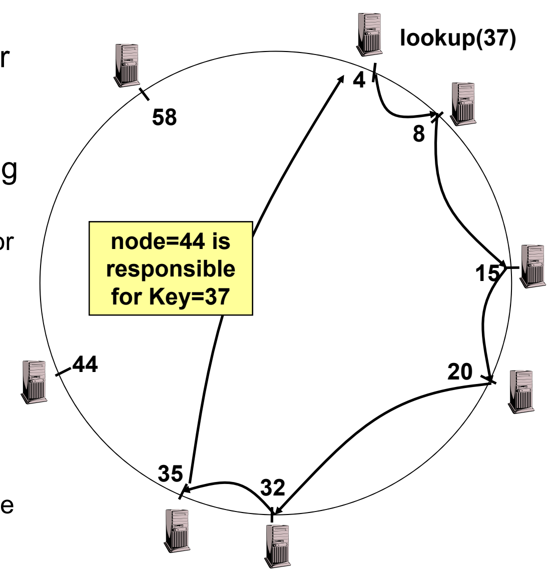


Chord: Distributed Lookup (Directory) Service

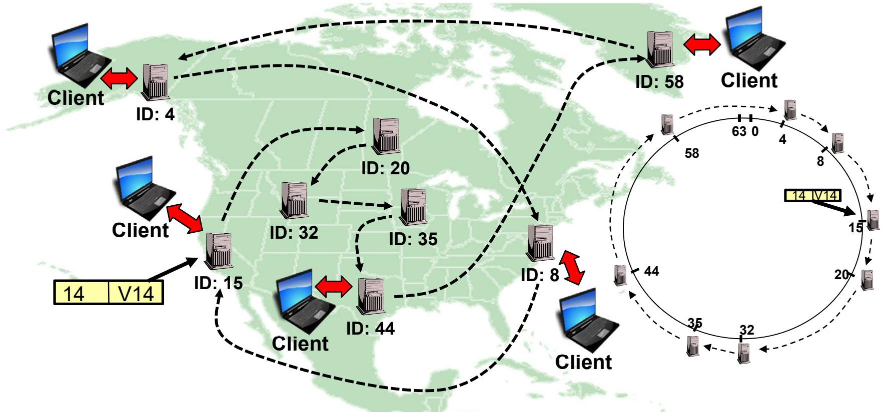
- "Chord" is a Distributed Lookup Service
  – Designed at MIT and here at Berkeley (Ion Stoica among others)
  – Simplest and cleanest algorithm for distributed storage
    » Serves as comparison point for other optims
- Import aspect of the design space:
  – Decouple correctness from efficiency
  – Combined *Directory* and *Storage*
- Properties
  – Correctness:
    » Each node needs to know about neighbors on ring (one predecessor and one successor)
    » Connected rings will perform their task correctly
  – Performance:
    » Each node needs to know about O(log(*M*)), where *M* is the total number of nodes
    » Guarantees that a tuple is found in O(log(*M*)) steps
- Many other *Structured, Peer-to-Peer* lookup services:
  – CAN, Tapestry, Pastry, Bamboo, Kademlia, …
  – Several designed here at Berkeley!

Chord's Lookup Mechanism: Routing!

- Each node maintains pointer to its successor
- Route packet (Key, Value) to the node responsible for ID using successor pointers
  – E.g., node=4 lookups for node responsible for Key=37
- Worst-case (correct) lookup is O(n)
  – But much better normal lookup time is O(log n)
  – Dynamic performance optimization (finger table mechanism)
    » More later!!!



But what does this really mean??



- Node names intentionally scrambled WRT geography!
  - Node IDs generated by secure hashes over metadata
    - » Including things like the IP address
  - This geographic scrambling spreads load and avoids hotspots
- Clients access distributed storage by accessing system through any member of the network

Summary (1/3)

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

Summary (2/3)

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

Summary (3/3)

- Key-Value Store:
  - Two operations
    - » put(key, value)
    - » value = get(key)
  - Challenges
    - » Scalability → serve get()'s in parallel; replicate/cache hot tuples
    - » Fault Tolerance → replication
    - » Consistency → quorum consensus to improve put() performance