Distributed File Systems

14B. Remote Data: Security
14C. Remote Data: Reliability & Robustness
14D. Remote Data: Performance
14E. Remote Data: Consistency
14F. Distributes Systems: Scalability

Security: Anonymous access

- all files available to all users
  - no authentication required
  - may be limited to read-only access
  - examples: anonymous FTP, HTTP
- advantages
  - simple implementation
- disadvantages
  - incapable of providing information privacy
  - write access often managed by other means

Peer-to-Peer Security

- client-side authentication/authorization
  - all users are known to all systems
  - all systems are trusted to enforce access control
  - example: basic NFS
- advantages
  - simple implementation
- limitations
  - assumes all client systems can be trusted
  - assumes all users are known to all systems
    - UID mapping between heterogeneous OSs
    - efficiency /scalability of universal user registries

Server Authenticated Sessions

- client agent authenticates to each server
  - session authorization based on those credentials
  - example: CIFS, authenticated HTTPS sessions
- advantages
  - simple implementation
- disadvantages
  - may not work in heterogeneous OS environment
  - universal user registry is not scalable
  - no automatic fail-over if server dies

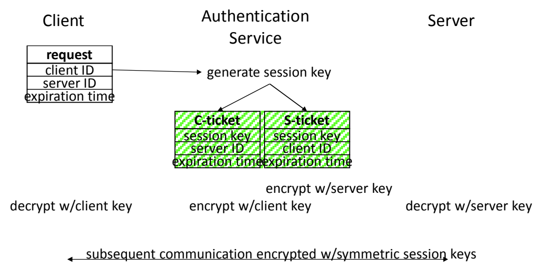
Domain Authentication Service

- independent authentication of client & server
  - each authenticates with authentication service
  - each knows/trusts only the authentication service
- authentication service issues signed “tickets”
  - assuring each of the others’ identity and rights
  - may be revocable or have a limited life-time
- may establish secure two-way session
  - privacy – nobody else can snoop on conversation
  - integrity – nobody can generate fake messages

example: KERBEROS

- establishes secure client/server sessions
- based on digital signatures
  - every agent has a secret (symmetric) key
  - keys are known only to agent, and KERBEROS
- request to KERBEROS encrypted w/client key
  - KERBEROS can decrypt it, authenticating requester
- KERBEROS response is two-part work ticket
  - part 1: encrypted with client's key
    - a symmetric session key
    - part 2 (to be forward, by client, to server)
  - part 2: encrypted with server's key
    - client ID, ticket duration,
    - symmetric session key

KERBEROS Work Tickets



Distributed Authorization

- Authentication service returns credentials
  - which server checks against Access Control List
  - advantage: auth service doesn't know about ACLs
- Authentication service returns Capabilities
  - which server can verify (by signature)
  - advantage: servers do not know about clients
- Both approaches are commonly used
  - credentials: if subsequent authorization required
  - capabilities: if access can be granted all-at-once
  - either may have an expiration time

Robustness: Embracing Failure

- Failures are inevitable
  - more components have more failures
  - complex systems have more modes of failure
  - we cannot build perfect components or systems
- We must build robust systems
  - additional capacity to survive failures
  - automatic failure detection
  - dynamically adapt to the new reality
  - continue service, despite component failures

Reliability and Availability

- Reliability … probability of not losing data
  - disk/server failures to not result in data loss
    - RAID (mirroring, parity, erasure coding)
    - copies on multiple servers
  - automatic recovery (of redundancy) after failure
- Availability … fraction of time service available
  - disk/server failures do not impact data availability
    - backup servers with automatic fail-over
  - automatic recovery (back up to date) after rejoin

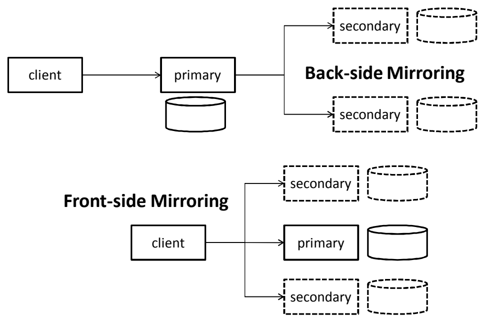
Problems and Solutions

- Network Errors – support client retries
  - RFS protocol uses idempotent requests
  - RFS protocol supports all-or-none transactions
- Client Failures – support server-side recovery
  - automatic back-out of uncommitted transactions
  - automatic expiration of timed out lock leases
- Server Failures – support server fail-over
  - replicated (parallel or back-up) servers
  - stateless RFS protocols
  - automatic client-server rebinding

Availability: Fail-Over

- data must be mirrored to secondary server
- failure of primary server must be detected
- client must be failed-over to secondary
- session state must be reestablished
  - client authentication/credentials
  - session parameters (e.g. working directory, offset)
- in-progress operations must be retransmitted
  - client must expect timeouts, retransmit requests
  - client responsible for writes until server ACKs

Reliability: Data Mirroring



Availability: Failure Detect/Rebind

- client driven recovery
  - client detects server failure (connection error)
  - client reconnects to (successor) server
  - client reestablishes session
- transparent failure recovery
  - system detects server failure (health monitoring)
  - successor assumes primary's IP address
  - state reestablishment
    - successor recovers last primary state check-point
    - stateless protocol

Availability: Stateless Protocols

- a statefull protocol (e.g. TCP)
  - operations occur within a context
  - each operation depends on previous operations
  - successor server must remember session state
- a stateless protocol (e.g. HTTP)
  - client supplies necessary context w/each request
  - each operation is complete and unambiguous
  - successor server has no memory of past events
- stateless protocols make fail-over easy

Availability: Idempotent Operations

- can be repeated many times with same effect
  - read block 100 of file X
  - write block 100 of file X with contents Y
  - delete file X version 3
  - non-idempotent operations
    - read next block of current file
    - append contents Y to end of file X
- if client gets no response, resend request
  - if server gets multiple requests, no harm done
  - works for server failure, lost request, lost response
    - but no ACK does not mean operation did not happen

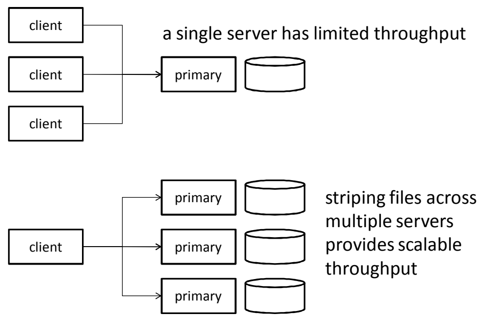
(nearly) Stateless Protocols

- client can maintain the session state
  - e.g. file handles and current offsets
- write operations can be made idempotent
  - e.g. associate a client XID with each write
- idempotence doesn't solve multi-writer races
  - competing writers must serialize their updates
  - clients cannot be trusted to maintain lock state
- we need a state-full Distributed Lock Manager
  - for whom failure recovery is extremely complex

Performance Challenges

- single client response-time
  - remote requests involve messages and delays
  - error detection/recovery further reduces efficiency
- aggregate bandwidth
  - each client puts message processing load on server
  - each client puts disk throughput load on server
  - each message loads server NIC and network
- WAN scale operation
  - where bandwidth is limited and latency is high
- aggregate capacity
  - how to transparently grow existing file systems

Performance: Bandwidth



Performance: Minimize Messaging

- Protocol features
  - as few messages as possible
  - client-side caching to eliminate read requests
  - aggregation for fewer/larger write requests
- Work Partitioning
  - do as much as possible on the client
  - do as much as possible on a single server
  - eliminate multi-node coordination
  - eliminate multi-node request forwarding

Performance: Read Requests

- client-side caching
  - eliminate waits for remote read requests
  - reduces network traffic
  - reduces per-client load on server
- whole file (vs. block) caching
  - higher network latency justifies whole file pulls
  - stored in local (cache-only) file system
  - satisfy early reads before entire file arrives
  - risk: may read data we won't actually use

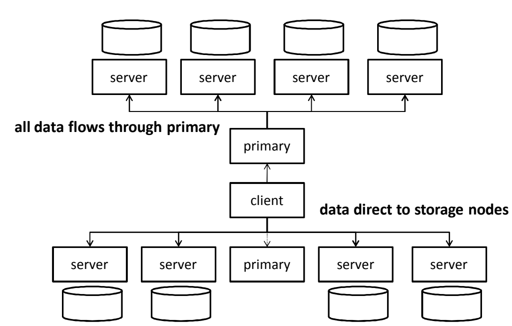
Performance: Write Requests

- write-back cache
  - create the illusion of fast writes
  - combine small writes into larger writes
  - fewer, larger network and disk writes
  - enable local read-after-write consistency
- whole-file updates
  - wait until *close(2)* or *fsync(2)*
  - reduce many successive updates to final result
  - possible file will be deleted before it is written
  - enable atomic updates, close-to-open consistency

Performance: Cost of Mirroring

- multi-host vs multi-disk mirroring
  - protects against host and disk failures
  - creates much additional network traffic
- mirroring by primary
  - primary becomes throughput bottleneck
  - replication traffic on back-side network
- mirroring by client
  - data flows directly from client to storage servers
  - replication traffic goes through client NIC
  - parity/erasure code computation on client CPU

Performance: Direct Data Path



(benefits of direct data path)

- architecture
  - primary tells clients where which data resides
  - client communicates directly w/storage servers
- throughput
  - data is striped across multiple storage servers
- latency
  - no intermediate relay through primary server
- scalability
  - fewer messages on network
  - much less data flowing through primary servers

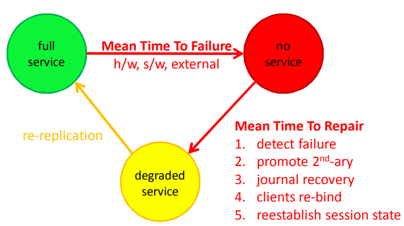
Performance: Partitioning the Work

open file instances, offsets data packing and unpacking	**clearly on client side**
authentication/authorization directory searching block caching	**either side (or both)**
logical to physical block mapping on-disk data representation device driver integration layer device driver	**clearly on server side**

Performance: Recovery Time

$$\textbf{Availability} = \frac{\textbf{MTTF}}{\textbf{MTTF + MTTR}}$$

we can try to maximize MTTF
we can try to minimize MTTR



(improving MTTR)

- MTTR (time before service can be restored)
  - primary failure detected (minimize)
  - secondary promoted to primary role (minimize)
  - recent/in-progress operations recovered
  - clients learn of change and re-bind
  - session state (if any) has been reestablished
- Degraded service may persist longer
  - restoring lost redundancy may take a while
  - heavily loading servers, disks, and network

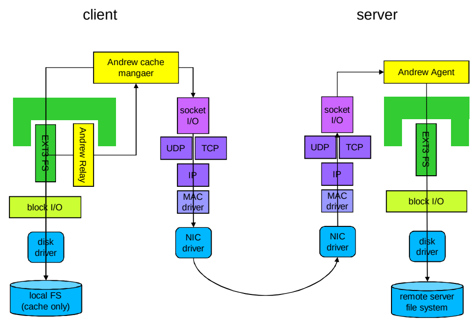
Performance: Cost of Consistency

- caching is essential in distributed systems
  - for both performance and scalability
- caching is easy in a single-writer system
  - force all writes to go through the cache
- multi-writer distributed caching is hard
  - Time To Live is a cute idea that doesn't work
  - constant validity checks defeat the purpose
  - one-writer-at-a-time is too restrictive for most FS
  - change notifications are a reasonable alternative

Andrew File System

- scalability, performance
  - large numbers of clients and very few servers
  - performance of local file systems
  - very low per-client load imposed on servers
  - no administration or back-up for client disks
- master files reside on a file server
  - local file system is used as a local cache
  - local reads satisfied from cache whenever possible
  - files are only read from server if not in cache
- simple synchronization of updates

Andrew File System Architecture



(Andrew File System – Replication)

- check for local copies in cache at open time
  - if no local copy exists, fetch it from server
  - if local copy exists, see if it is still up-to-date
    - compare file size and modification time with server
  - optimizations reduce overhead of checking
    - subscribe/broadcast change notifications
    - time-to-live on cached file attributes and contents
- send updates to server when file is closed
  - wait for all changes to be completed
  - file may be deleted before it is closed

Andrew File System – Reconciliation

- updates sent to server when local copy closed
- server notifies all clients of change
  - warns them to invalidate their local copy
  - warns them of potential write conflicts
- server supports only advisory file locking
  - distributed file locking is extremely complex
- clients are expected to handle conflicts
  - noticing updates to files open for write access
  - notification/reconciliation strategy is unspecified

Rating Andrew File System

- Performance and Scalability
  - all file access by user/applications is local
  - update checking (with call-backs) is relatively cheap
  - both fetch and update propagation are very efficient
  - minimal per-client server load (once cache filled)
- Robustness
  - no server fail-over, but have local copies of most files
- Transparency
  - mostly perfect - all file access operations are local
  - pray that we don't have any update conflicts

Andrew File System vs. NFS

- design centers
  - both designed for continuous connection client/server
  - NFS supports diskless clients w/o local file systems
- performance
  - AFS generates much less network traffic, server load
  - they yield similar client response times
- ease of use
  - NFS provides for better transparency
  - NFS has enforced locking and limited fail-over
- NFS requires more support in operating system

Complication: Failure & Rejoin

- a file server goes down
  - no problem another server handles his clients
- then he comes back up and reports for work
  - he needs to get all the updates he missed
- How do we know what updates he missed?
  - we could compare all of his files with all of ours
    - that could take a very long time
  - we can keep a log of all recent updates
    - but we have to know which ones he already has
    - maybe files are versioned, or updates are numbered

Complication: Split-Brain

- suppose we had a network failure
  - that partitioned our file servers
  - and each half tried to take over for the other
  - and each half processed different write operations
- How could we reconcile the changes
  - we could merge updated versions of different files
  - what about files that were changed in both halves?
- Quorum rules can prevent "dueling servers"
  - servers that can't make quorum are read-only

Complication: Disconnected Operation

- Consider a notebook and a file server
  - I synchronize my notebook with the file server
  - I go away on a trip and update many files
  - others may change the same files on the server
- How can we identify all of the changes?
  - Intercept & log all changes (e.g. Windows Briefcase)
  - Differential Analysis vs. a baseline (e.g. rsync)
- How can we correctly reconcile conflicts?
  - perhaps some can be handled automatically
  - some may require manual (human) resolution

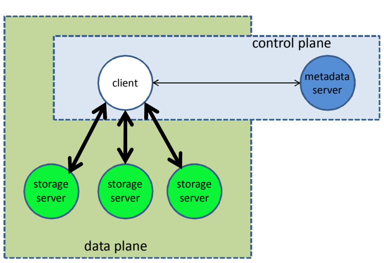
Scalability – Traffic

- network messages are expensive
  - NIC and network capacity to carry them
  - server CPU cycles to process them
  - client delays awaiting responses
- minimize messages/client/second
  - cache results to eliminate requests entirely
  - enable complex operations w/single request
  - buffer up large writes in write-back cache
  - pre-fetch large reads into local cache

Scalability - Bottlenecks

- avoid a single control points
  - partition responsibility over many nodes
- separated data- and control-planes
  - control nodes choreograph the flow of data
    - where data should be stored or obtained from
    - ensuring coherency and correct serialization
  - data flows directly from producer to consumer
    - data paths are optimized for throughput/efficiency
- dynamic re-partitioning of responsibilities
  - in response to failures and/or load changes

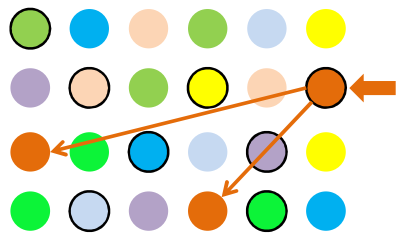
Control and Data Planes



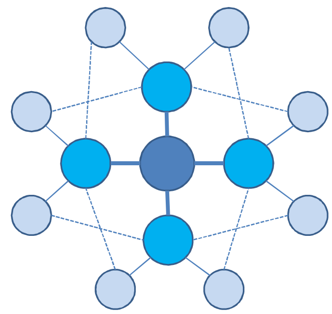
Scalability: Cluster Protocols

- Consensus protocols do not scale well
  - they only work for small numbers of nodes
- Minimize number of consensus operations
  - elect a single master who makes decisions
  - partitioned and delegated responsibility
- Avoid large-consensus/transaction groups
  - partition work among numerous small groups
- Avoid high communications fan-in/fan-out
  - hierarchical information gathering/distribution

Data Plane: small transaction clusters



Control Plane: hierarchical reporting



Assignments

- Projects
  - get started on P4C
    - SSL connections may be difficult to debug
    - there are no slip days on this project
- Reading (31pp)
  - AD C10 (SMP scheduling)
  - Multi-Processors
  - Clustering Concepts
  - Horizontally Scaled Systems
  - Eventual Consistency
  - AD appx B (virtual machines)