Network and Distributed File Systems
CS 111
Operating System Principles
Peter Reiher

Outline

- Goals and challenges of providing file systems over the network
- Basic architectures
- Major issues
  - Authentication and security
  - Performance
- Examples of networked file systems

Network File Systems: Goals and Challenges

- Sometimes the files we want aren't on our machine
- We'd like to be able to access them anyway
- How do we provide access to remote files?
  - Basic goals
  - Functionality challenges
  - Performance challenges
  - Robustness challenges
  - Manageability challenges

Basic Goals

- Transparency
  - Indistinguishable from local files for all uses
  - All clients see all files from anywhere
- Performance
  - Per-client: at least as fast as local disk
  - Scalability: unaffected by the number of clients
- Cost
  - Capital: less than local (per client) disk storage
  - Operational: zero, it requires no administration
- Capacity: unlimited, it is never full
- Availability: 100%, no failures or service down-time

Functionality Challenges

- Transparency
  - Making remote files look just like local files
    - On a network of heterogenous clients and servers
    - In the face of Deutch's warnings
  - Creating global file name-spaces
- Security
  - WAN scale authentication and authorization
- Providing ACID properties
  - Atomicity, Consistency, Isolation, Durability

Performance Challenges

- Single client response-time
  - Remote requests involve messages and delays
- Aggregate bandwidth
  - Each client puts message processing load on server
  - Each client puts disk throughput load on server
  - Each message loads server's NIC and network
- WAN scale operation
  - Where bandwidth is limited and latency is high
- Aggregate capacity
  - How to transparently grow existing file systems

Robustness Challenges

- All files should always be available, despite …
  - Failures of the disk on which they are stored
  - Failures of the remote file server
  - Regional catastrophes (flood, earthquake, etc.)
  - Users having deleted the files
- Fail-over should be prompt and seamless
  - A delay of a few seconds <u>might</u> be acceptable
- Recovery must be entirely automated
  - For time, cost, and correctness reasons

Manageability Challenges

- Storage management
  - Integrating new storage into the system
  - Diagnosing and replacing failed components
- Load and capacity balancing
  - Spreading files among volumes and servers
  - Spreading clients among servers
- Information life cycle management
  - Moving unused files to less expensive storage
  - Archival “compliance,” finding archived data
- Client configuration
  - Domain services, file servers, name-spaces, authentication

Security Challenges

- What meaningful security can we provide for networked file systems?
- Can we guarantee reasonable access control?
- How about secrecy of data crossing the network?
- How can we provide integrity guarantees to remote users?
- What if we can't trust all of the systems requesting files?
- What if we can't trust all of the systems storing files?

Key Characteristics of Network File System Solutions

- APIs and transparency
  - How do users and processes access remote files?
  - How closely do remote files mimic local files?
- Performance and robustness
  - Are remote files as fast and reliable as local ones?
- Architecture
  - How is solution integrated into clients and servers?
- Protocol and work partitioning
  - How do client and server cooperate?

Remote File Systems

- The simplest form of networked file system
- Basically, going to a remote machine to fetch files
- Perhaps with some degree of abstraction to hide unpleasant details
- But generally with a relatively low degree of transparency
  - Remote files are obviously remote

Explicit File Copying

- User-invoked commands to transfer files
  - Copy from remote to local site, then use as a local file
- Typical architecture
  - Client-side: interactive command line interface
    - May include powerful features like wild-cards, multi-file transfer, scheduled delivery, automatic difference detection, GUIs, etc.
  - Server-side: user mode, per client daemon
    - Basically, only this daemon knows file access is remote
- Many protocols are IETF standards
  - Early approaches simple and general (FTP, TFTP)
  - Modern approaches often cloud based (Dropbox, Amazon Web Services, etc.)

Advantages and Disadvantages

- Advantages
  - User-mode client/server implementations
  - Efficient transfers (fast and with little overhead)
  - User directly controls what is transferred when
  - Modern versions integrate with local OS
- Disadvantages
  - Local and remote files are totally different
  - Manual transfers are tedious and error prone
- Contemporary usage
  - Sharing files across the Internet
  - Mass storage/backup from local machine

Remote Access Methods

- Distinct APIs for accessing remote files
  - Standard open/close/read/write are "local only"
  - Use different routines to access remote files
- Distinct user interface for remote files
  - Use a browser instead of a shell or finder
- User-mode implementation
  - Client remote access library, browser command
  - Protocols and servers similar to rcp/FTP
- New file naming schemes (e.g., URLs)

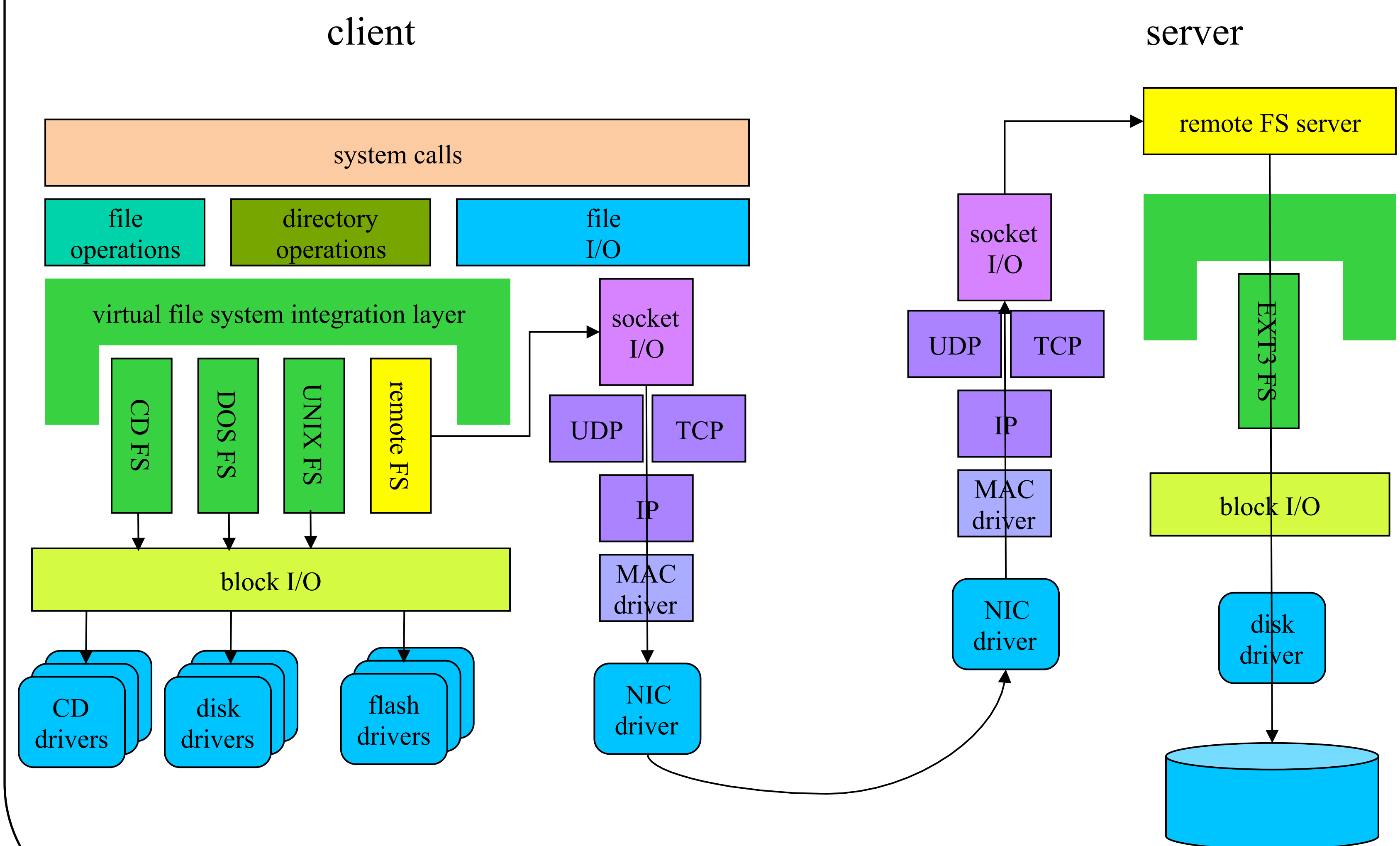
Advantages and Disadvantages

- Advantages
  - User-mode client/server implementations
  - Services can be designed to suit modes of file use
  - Services encapsulate location of actual data
- Disadvantages
  - Only works for a few programs (e.g., browsers)
  - All other programs (e.g., editors) are "local only"
  - Local and remote files pretty distinct
  - Often no support for writing (or a special interface)
- Contemporary usage
  - Many key applications: browsers, e-mail, SQL

Remote File Access Protocols

- Goal: complete transparency
  - Normal file system calls work on remote files
  - Support file sharing by multiple clients
  - High performance, availability, reliability, scalability
- Typical Architecture
  - Uses plug-in file system architecture
  - Client-side file system is merely a local proxy
  - Translates file operations into network requests
  - Server-side daemon receives/process requests
  - Translates them into real file system operations

Remote File Access Architecture



The Client Side

- On Unix/Linux, makes use of VFS interface
- Allows plug-in of file system implementations
  – Each implements a set of basic methods
    - create, delete, open, close, link, unlink, etc.
  – Translates logical operations into disk operations
- Remote file systems can also be implemented
  – Translate each standard method into messages
  – Forward those requests to a remote file server
  – RFS client only knows the RFS protocol
    - Need not know the underlying on-disk implementation

Server Side Implementation

- RFS Server Daemon
  - Receives and decodes messages
  - Does requested operations on local file system
- Can be implemented in user- or kernel-mode
  - Kernel daemon may offer better performance
  - User-mode is much easier to implement
- One daemon may serve all incoming requests
  - Higher performance, fewer context switches
- Or could be many per-user-session daemons
  - Simpler, and probably more secure

Advantages and Disadvantages

- Advantages
  - Very good application level transparency
  - Very good functional encapsulation
  - Able to support multi-client file sharing
  - Potential for good performance and robustness
- Disadvantages
  - At least part of implementation must be in the OS
  - Client and server sides tend to be fairly complex
- Contemporary use
  - Ubiquitous today

Clustered File Servers

- Use several cooperating file servers in one of the previously discussed ways
- Can aggregate their bandwidth and storage capacity
- Allows client load and file capacity balancing
- Virtualized storage cluster allows us to respond to difficult customer demands
  - Infinite bandwidth
  - Capacity scalability
  - Minimal down-time

Degrees of Distribution

- Remote file access
  - One server owns disks and implements file systems
  - Clients access files via remote access protocols
- Clustered file servers
  - Multiple servers, each owns disks and file systems
  - Cooperate to provide a single virtual NAS service
- Distributed file systems
  - N servers and M disks
  - Multiple servers can concurrently use same disk
  - "Don't try this one at home, kids"

Remote File Access: Problems and Solutions

- Authentication and authorization
- Performance
- Synchronization
- Robustness

Authorization and Authentication

- Authorization is determined if someone is allowed to do something
- Authentication is determining who someone is
- Both are required for good file system security
  - Be sure who someone is first
  - Then see if that entity is allowed to do what he asked for
- Both are more challenging when file system spans multiple machines

Problems in Authentication/ Authorization

- How does remote server know requestor identity?
  - User isn’t logged into his machine
- Where should we enforce access control rules?
  - On the requesting client side?
    - That’s who really knows who the client is
  - On the responding server side?
    - That’s who has responsibility to protect the data
  - On both?
- Name space issues
  - Do the client and server agree on who’s who?

Approaches to These Security Issues

- User-session protocols (e.g., CIFS)
  - RFS session establishment includes authentication
    - So server authenticates requesting client
  - Server performs all authorization checks
- Peer-to-peer protocols (e.g., NFS)
  - Server trusts client to enforce authorization control
  - And to authenticate the user
- Third party authentication (e.g., Kerberos)
  - Server checks authorization based on credentials

Performance Issues

- Performance of the remote file system now dependent on many more factors
  - Not just the local CPU, bus, memory, and disk
- Also on the same hardware on the server that stores the files
  - Which often is servicing many clients
- And on the network in between
  - Which can have wide or narrow bandwidth

Some Performance Solutions

- Appropriate transport and session protocols
  - Minimize messages, maximize throughput
- Partition the work
  - Minimize number of remote requests
  - Spread load over more processors and disks
- Client-side pre-fetching and caching
  - Fetching whole file at a once is more efficient
  - Block caching for read-ahead and deferred writes
  - Reduces disk I/O and network I/O (vs. server cache)

Protocol-Related Solutions

- Minimize messages
  - Allow any key operation to be performed with a single request and a single response
  - Combine short messages and responses into a single packet
- Maximize throughput
  - Design for large data transfers per message
  - Use minimal flow control between client and server

Partitioning the Work

Open file instances, offsets

Data packing and unpacking

Clearly on client side

Authentication/authorization

Directory searching

Block caching

Specialized caching (directories, file descriptors)

Either side (or both)

Logical to physical block mapping

On-disk data representation

Device driver integration layer

Device driver

Clearly on server side

Server Load Balancing

- If multiple servers can handle the same file requests, we can load balance
  - Improving performance for multiple clients
- Provide a pool of servers
  - All with access to the same data
    - E.g., they all have copies of all the same files
  - Spread client traffic across all of the servers
    - E.g., using a load-balancing front-end router
  - Increase capacity by adding servers to pool
    - With potentially linear scalability
  - Works best if requests are idempotent

Client-Side Caching

- Benefits
  - Avoids network latencies
  - Clients can cache name-to-handle bindings
    - Eliminating repetition of the same search
  - Clients can cache blocks of file data
    - Eliminating the need to re-fetch them from the server
- Dangers
  - Multiple clients, each with his own cache
  - Cache invalidation issues
- Challenges
  - Serializing concurrent writes from multiple clients
  - Keeping client side caches up-to date
    - Without sending N messages per update

The Cache Invalidation Issue

- Two (or more) clients cache the same block
- One of them updates it
- What about the other one?
- Server could notify every client of every write
  - Very inefficient
- Server could track which clients to notify
  - Higher server overhead
- Clients could obtain lock on files before update
- Clients could verify cache validity before use

Synchronization Issues

- Distributed synchronization is slow and difficult
  - Provide a centralized synchronization server
    - All locks are granted by a single server
    - Changes are not official until he acknowledges them
    - He notifies other nodes of “interesting” changes
- Distributed systems have complex failure modes
  - Locks are granted as revocable leases
    - Update transaction must be accompanied by valid lease
  - Versioned files can detect stale information
  - All cached information should have a “time to live”
    - A tradeoff between performance and consistency

Robustness Issues

- Three major components in remote file system operations
  - The client machine
  - The server machine
  - The network in between
- All can fail
  - Leading to potential problems for the remote file system's data and users

Robustness Solution Approaches

- Network errors – support client retries
  - Have file system protocol uses idempotent requests
  - Have protocol support 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 remote file system protocols
  - Automatic client-server rebinding

Idempotent Operations

- Operations that can be repeated many times with same effect as if done once
  - If server does not respond, client repeats request
  - If server gets request multiple times, no harm done
- Examples:
  - Read block 100 of file X
  - Write block 100 of file X with contents Y
  - Delete file X, version v
- Examples of non-idempotent operations:
  - Read next block of current file
  - Append contents Y to end of file X

State-full and Stateless Protocols

- A state-full protocol has a notion of a "session"
  - Context for a sequence of operations
  - Each operation depends on previous operations
  - Server is expected to remember session state
  - Examples: TCP (message sequence numbers)
- A stateless protocol does not assume server retains "session state"
  - Client supplies necessary context on each request
  - Each operation is complete and unambiguous
  - Example: HTTP

Server Fail-Over

- When is handling server failure by switching to another server feasible?
  - If the other server can access the required data
    - Because files are replicated to multiple servers
    - Because new server can access old server's disks
  - If the protocol allows stateless servers
    - Client will not expect server to remember anything
  - If clients can be re-bound to a new server
    - IP address fail-over may make this automatic
    - RFS client layer might rebind w/o telling application
    - Idempotent requests can be re-sent with no danger

Remote File System Examples

- Common Internet File System (classic client/ server)
- Network File System (peer-to-peer file sharing)
- Andrew File System (cache-only clients)
- Hyper-Text Transfer Protocol (a different approach)

Common Internet File System

- Originally a proprietary Microsoft Protocol
  - Newer versions (CIFS 1.0) are IETF standard
- Designed to enable "work group" computing
  - Group of PCs sharing same data, printers
  - Any PC can export its resources to the group
  - Work group is the union of those resources
- Designed for PC clients and NT servers
  - Originally designed for FAT and NT file systems
  - Now supports clients and servers of all types

CIFS Architecture

- Standard remote file access architecture
- State-full per-user client/server sessions
  - Password or challenge/response authentication
  - Server tracks open files, offsets, updates
  - Makes server fail-over much more difficult
- Opportunistic locking
  - Client can cache file if nobody else using/writing it
  - Otherwise all reads/writes must be synchronous
- Servers regularly advertise what they export
  - Enabling clients to "browse" the workgroup

Benefits of Opportunistic Locking

- A big performance win
- Getting permission from server before each write is a huge expense
  - In both time and server loading
- If no conflicting file use 99.99% of the time, opportunistic locks greatly reduce overhead
- When they can’t be used, CIFS does provide correct centralized serialization

CIFS Pros and Cons

- Performance/Scalability
  - Opportunistic locks enable good performance
  - Otherwise, forced synchronous I/O is slow
- Transparency
  - Very good, especially the global name space
- Conflict prevention
  - File/record locking and synchronous writes work well
- Robustness
  - State-full servers make seamless fail-over impossible

The Network File System (NFS)

- Transparent, heterogeneous file system sharing
  - Local and remote files are indistinguishable
- Peer-to-peer and client-server sharing
  - Disk-full clients can export file systems to others
  - Able to support diskless (or dataless) clients
  - Minimal client-side administration
- High efficiency and high availability
  - Read performance competitive with local disks
  - Scalable to huge numbers of clients
  - Seamless fail-over for all readers and some writers

The NFS Protocol

- Relies on idempotent operations and stateless server
  - Built on top of a remote procedure call protocol
  - With eXternal Data Representation (XDR), server binding
  - Versions of RPC over both TCP or UDP
  - Optional encryption (may be provided at lower level)
- Scope – basic file operations only
  - Lookup (open), read, write, read-directory, stat
  - Supports client or server-side authentication
  - Supports client-side caching of file contents
  - Locking and auto-mounting done with another protocol

NFS Authentication

- How can we trust NSF clients to authenticate themselves?
- NFS not not designed for direct use by user applications
- It permits one operating system instance to access files belonging to another OS instance
- If we trust the remote OS to see the files, might as well trust it to authenticate the user
- Obviously, don't use NFS if you don't trust the remote OS . . .

NFS Replication

- NFS file systems can be replicated
  - Improves read performance and availability
  - Only one replica can be written to
- Client-side agent (in OS) handles fail-over
  - Detects server failure, rebinds to new server
- Limited transparency for server failures
  - Most readers will not notice failure (only brief delay)
  - Users of changed files may get "stale handle" error
  - Active locks may have to be re-obtained

NFS and Updates

- An NFS server does not prevent conflicting updates
  - As with local file systems, this is application's job
- Auxiliary server/protocol for file and record locking
  - All leases are maintained on the lock server
  - All lock/unlock operations handed by lock server
- Client/network failure handling
  - Server can break locks if client dies or times out
  - "Stale-handle" errors inform client of broken lock
  - Client response to these errors are application specific
- Lock server failure handling is very complex

NFS Pros and Cons

- Transparency/Heterogeneity
  - Local/remote transparency is excellent
  - NFS works with all major OSes and FSes
- Performance
  - Read performance may be better than local disk
  - Replication provides scalable read bandwidth
  - Write performance slower than local disk
- Robustness
  - Transparent fail-over capability for readers
  - Recoverable fail-over capability for writers

NFS Vs. CIFS

- Functionality
  - NFS is much more portable (platforms, OS, FS)
  - CIFS provides much better write serialization
- Performance and robustness
  - NFS provides much greater read scalability
  - NFS has much better fail-over characteristics
- Security
  - NFS supports more security models
  - CIFS gives the server better authorization control

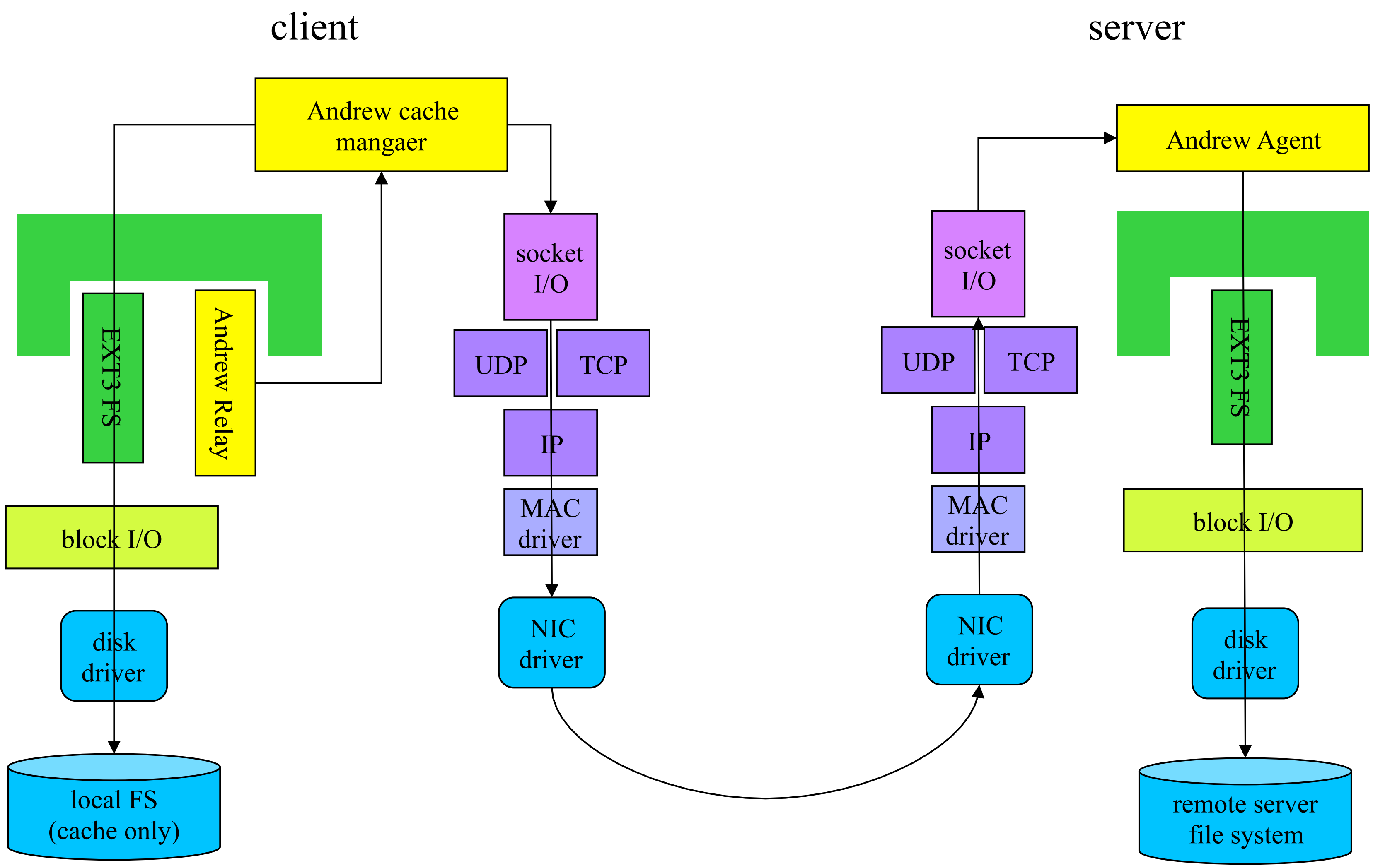
The Andrew File System

- AFS
- Developed at CMU
- Designed originally to support student and faculty use
  - Generally, large numbers of users of a single organization
- Uses a client/server model
- Makes use of whole-file caching

AFS Basics

- Designed for scalability, performance
  - Large numbers of clients and very few servers
  - Needed 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 when possible
  - Files are only read from server if not in cache
- Simple synchronization of updates

AFS Architecture



AFS Replication

- One replica at server, possibly many at clients
- 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
    - E.g., temporary files that servers need not know about

AFS Reconciliation

- Client sends updates 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

AFS Pros and Cons

- Performance and Scalability
  - All file access by user/applications is local
  - Update checking (with time-to-live) 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

AFS vs. NFS

- Basic designs
  - Both designed for continuous connection client/server
  - NFS supports diskless clients without 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

HTTP

- A different approach, for a different purpose
- Stateless protocol with idempotent operations
  - Implemented atop TCP (or other reliable transport)
  - Whole file transport (not remote data access)
    - **get** file, **put** file, **delete** file, **post** form-contents
  - Anonymous file access, but secure (SSL) transfers
  - Keep-alive sessions (for performance only)
- A truly global file namespace (URLs)
  - Client and in-network caching to reduce server load
  - A wide range of client redirection options

HTTP Architecture

- Not a traditional remote file access mechanism
- We do not try to make it look like local file access
  - Apps are written to HTTP or other web-aware APIs
  - No interception and translation of local file operations
  - But URLs can be constructed for local files
- Server is entirely implemented in user-mode
  - Authentication via SSL or higher level dialogs
  - All data is assumed readable by all clients
- HTTP servers provide more than remote file access
  - POST operations invoke server-side processing
- No attempt to provide write locking or serialization

HTTP Pros and Cons

- Transparency
  - Universal namespace for heterogeneous data
  - Requires use of new APIs and namespace
  - No attempt at compatibility with old semantics
- Performance
  - Simple implementations, efficient transport
  - Unlimited read throughput scalability
  - Excellent caching and load balancing
- Robustness
  - Automatic retrys, seamless fail-over, easy redirects
  - Not much attempt to handle issues related to writes

HTTP vs. NFS/CIFS

- The file model and services provided by HTTP are much weaker than those provided by CIFS or NFS
- So why would anyone choose to use HTTP for remote file access?
- It's easy to use, provides excellent performance, scalability and availability, and is ubiquitous
- If I don't need per-user authorization, walk-able name spaces, and synchronized updates,
  - Why pay the costs of more elaborate protocols?
  - If I do need, them, though, . . .

Conclusion

- Be clear about your remote file system requirements
  - Different priorities lead to different tradeoffs & designs
- The remote file access protocol is the key
  - It determines the performance and robustness
  - It imposes or presumes security mechanisms
  - It is designed around synchronization & fail-over mechanisms
- Stateless protocols with idempotent ops are limiting
  - But very rewarding if you can accept those limitations
- Read-only content is a pleasure to work with
  - Synchronized and replicated updates are very hard