Distributed Systems

Monsoon 2024

Lecture 7

International Institute of Information Technology

Hyderabad, India

Most slides taken from the GFS conference talk at ACM SOSP 2003. Slides on NFS and AFS are developed from the lecture notes of Roxana Geambasu, Columbia U.

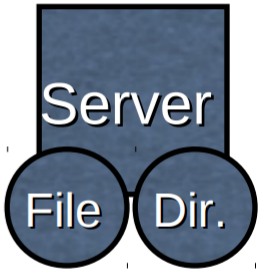
What is a FileSystem?

- Provides support for access and also access control.
- The UNIX file system has a mechanism based on inodes.
  - Direct and indirect numbering
  - Access control based on owner/ group/ others, for read/ write/ execute.
  - Formed the basis for many other file system designs.
- For more details, we refer the reader to the book by Bach.

Read a few chapters of the book by Maurice J Bach on the Design of the Unix Operating System

What is a File System?

File Ops	Directory Ops
Open	Create file
Read	Mkdir
Write	Rename file
Close	Rename directory
Seek	Delete file
	Delete directory



Current Trends

- Emergence of planet-scale applications such as Facebook and YouTube.
- These applications allow multiple users to create, share, and access content simultaneously.
- The amount of content generated each day is of the order of petabytes or more.
- Most of the data and the users of the data are more likely to be geographically separated yet seamlessly connected.
- Data is the new oil – source of analytics, revenue, systems, …

Current Trends

- To benefit from the data, this data has to be stored in appropriate manner, including crossing hurdles such as
  - Scalability
  - Retrieval
  - Network latency
  - Concurrent access
  - Availability
  - Durability
  - And so on

Design Features for Distributed File Systems

- Durability
  - The ability to recover data in case of failures to the underlying hardware.
  - Distributed file systems usually employ a form of replication to safeguard against such issues.
  - Replication however comes with a set of challenges involving the creation/ updation/ maintenance/ and consistency of the replicas.

Design Features for Distributed File Systems

- Availability
  - Distributed file systems have to contend with possible errors from the users or the file system itself.
  - Delays and other network induced vagaries affect the file system and its operations

Design Features for Distributed File Systems

- Consistency Model
  - Multiple users create/ read/ update/ write to the same set of files, possibly at the same time.
  - Think of systems such as dropbox, google docs, OneDrive etc.
  - Need mechanisms to support such concurrent updates
  - Semantics and guarantees on what happens to concurrent updates

Design Features for Distributed File Systems

- Scalability
  - Provide for scale as the storage requirements grow
  - Allow for scaling up the capacity in a seamless and transparent manner without degrading performance.

Design Features for Distributed File Systems

- Metadata
  - Namespace and the associated data structures form the bulk of how a file system maintains files.
  - Understand the memory system considerations while using the data structures of choice
  - These choices impact latency and throughput.

Design Features for Distributed File Systems

- Other Application Specific Concerns
  - What are the typical use cases?
    - Helps understand system features and requirements
  - What are the typical file sizes?
    - Helps decide on parameters such as block size.
  - What functionality or functionalities are more important from a performance point of view?
    - To support better throughput, may want to optimize on key functionalities even at the expense of some others.

The Design Space

- Now that we have identified a few issues, let us understand the potential choices for system designers.

The Design Space

- Flat vs Hierarchical namespace
  - Flat namespace requires each file to be given a unique identifier.
  - These identifiers are used to locate the required file.
  - Hierarchical namespace allows for a pathname based identification.

The Design Space

- How the read/write are supported?
- Some of the options are
  - A fully client-server system
  - Server hands off to client.
  - Server helps client locate the file
  - Server allows the client to cache the file locally either in full or in parts.
    - Can create inconsistent views.

The Design Space

- Guarantees on consistency has a full spectrum of possibilities.
  - Leave to the clients: no guarantees offered.
  - Forbid concurrent updates. Not very useful
  - Last Write Wins: Serialized view of writes, and the last successful write is what the server keeps

The Design Space

- Application specific concerns
  - Combine multiple objects into a single file
    - Photos in facebook
  - Disallow write/update and create a new copy on update.
  - Support operations such as append instead of write.
    - Log files

The Design Space

- Handling faults
  - Provide for multiple servers
  - Replication of data and its associated challenges

Distributed File Systems

- Earliest versions of distributed file systems:
  - NFS – Networked File System
    - Developed at Sun Microsystems which believed in the slogan “The Network is the Computer”
  - AFS – Andrew File System
    - Part of the Andrew project at CMU that created a distributed computing environment at CMU.
- Modern projects on distributed file systems include
  - GFS – The Google File System,
  - HDFS – the OpenSource version of GFS
  - Colossus – the next version of GFS
  - Haystack, Tectonic – the systems used by Facebook
- Among all, there are some common notions that we will explore today.

Goals of NFS and AFS

- Typical goals of a file system:
  - Have a consistent namespace for files
  - Let authorized users access their files and perform file operations
- On the other hand, distributed systems and distributed file systems have a variety of goals including
  - Scalability: Support large user base and large file base
  - Fault tolerance: Tolerate client and server failures
  - Concurrency: Allow multiple users
  - Security: Authorized users only
  - ...
- Cannot meet all goals all the time
  - Prioritize towards most important goals.

Design Principle of Distributed File Systems

- User-Oriented Vs. Workload-oriented design
  - Workload oriented design measures the characteristics of target workloads to inform the design
  - On the other hand, user-oriented designs optimize the system towards how users use files.
- User-oriented designs optimize to how users use files (vs. big programs)
  - Most files are privately owned
  - Not too much concurrent access
  - Sequential access is common; reads more common than writes
  - Example: AFS and NFS
- Distributed file systems such as GFS are geared towards big-program/big-data workloads

Early Design Options

- Use RPC to forward every FS operation to the server
  - Server orders all accesses, performs them, and sends back result
  - NFS v1 was built like this.

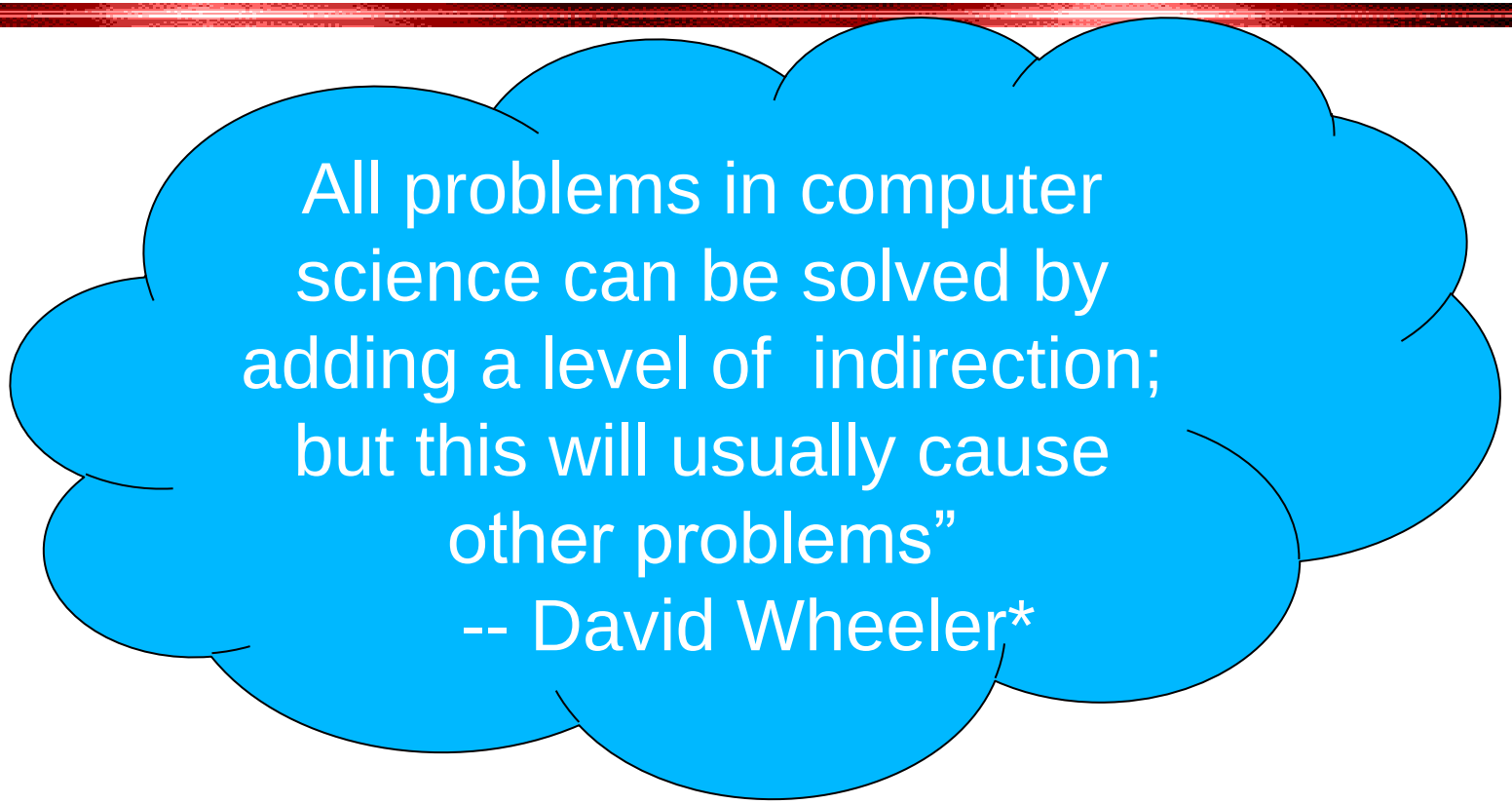
Early Design Options

- Use RPC to forward every FS operation to the server
  - Server orders all accesses, performs them, and sends back result
  - NFS v1 was built like this.
- Plus: Same behavior as if both programs were running on the same local filesystem!
- Minuses:
  - Performance will suffer. Latency of access to remote server often much higher than to local memory.
  - Moreover, server would get overloaded as accessing server is needed for every action.

Early Design Options

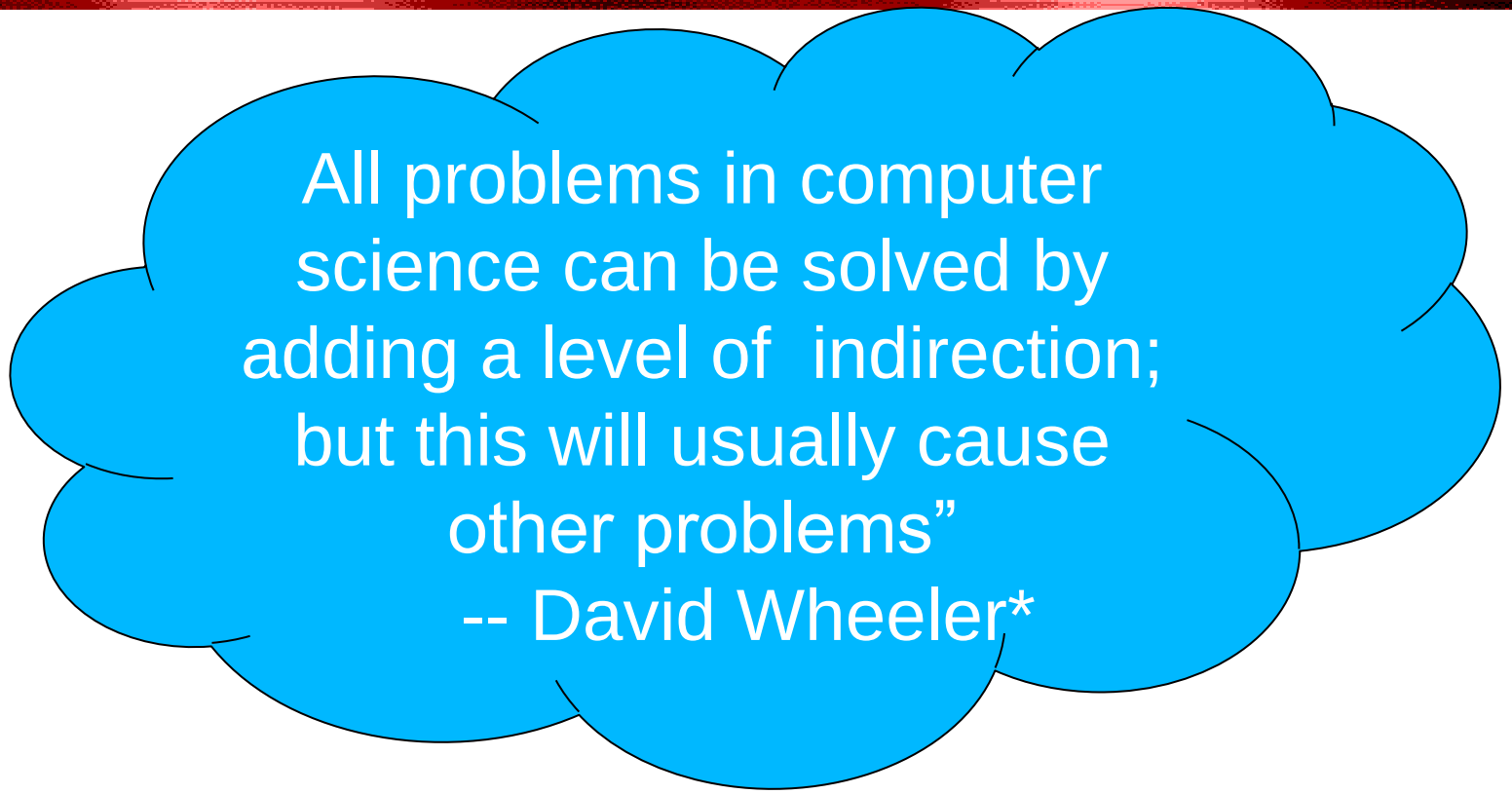
- Use RPC to forward every FS operation to the server
  - Server orders all accesses, performs them, and sends back result
  - NFS v1 was built like this.
- Plus: Same behavior as if all programs were running on the same local filesystem!
- Minuses:
  - Performance will suffer. Latency of access to remote server often much higher than to local memory.
    - Moreover server would get overloaded as accessing server is needed for every action.
- Moral: We need designs that avoid accessing the server for everything? What can we avoid this for? What do we lose in the process?

A Better solution



- Per above, we will try the same idea for distributed file systems along with caching.
- But we should understand the risks of caching in terms of consistency.

A Better solution



- Per above, we will try the same idea for distributed file systems along with caching.
- But we should understand the risks of caching in terms of consistency.

* of the Burrows-Wheeler Transform used in data compression

NFS

- Cache file blocks, directory metadata in RAM at both clients and servers.
- Plus: No network traffic if open/read/write/close can be done locally.
- Minus: failures and cache consistency are big concerns with this approach
- NFS trades some consistency for increased performance...

Problems with Caching

- Server crashes
  - Any data that's in memory but not on disk is lost
  - What if client does seek(); /* SERVER CRASH */; read()
    - If server maintains file position in RAM, the read will return bogus data
- Lost messages
  - What if a client loses an acknowledgement from the server for `delete("foo")`
  - And in the meantime, another client created the file with the same name anew?
  - The first client might retry the delete and delete new file
- Client crashes
  - Might lose data updates in client cache

Stateful Vs Stateless Protocols

- Stateful Protocols: Server maintains client-specific state
  - Shorter requests
  - Better performance in processing requests
  - Cache coherence is possible – Server can know who's accessing what
  - File locking is possible

- Stateless Protocols: Server maintains no information on client accesses
  - Each request must identify file and offsets
  - Server can crash and recover – No state to lose
  - Client can crash and recover
  - No open/close needed
    - They only establish state
    - No server space used for state – can support many clients
    - File locking not possible

NFS Solution

- Stateless design
  - Flush-on-close: When file is closed, all modified blocks sent to server. close() does not return until bytes safely stored.
- Stateless protocol: requests specify exact state.
  - read() -> read([position]). no seek on server.
- Operations are idempotent
  - How can we ensure this? Unique IDs on files/directories.
  - It's not delete("foo"), it's delete(1337f00f), where that ID won't be reused.
    - See the level of indirection

Caching and lack of Consistency

- A writer and a reader may notice inconsistency due to writes not taking effect at the server but only in the cache.
- NFS allows for this inconsistency.
- Requires flush on close.
  - Flush all updates from cache to server on close operation.
- This means the system can be inconsistent for a few seconds
  - Two clients doing a read() at the same time for the same file could see different results if one had old data cached and the other didn't.
- Periodic checks to minimize damage.
- Called as weak consistency.
- NFS provides no guarantees at all on multiple writes.

Later Versions of NFS

- Beyond NFS v1.0, there were NFS versions up to NFS v4.0 developed over the years.
- NFS v3.0 introduced the asynchronous write operation.
  - In the semantics of asynchronous write operation, the client sends the write data to the server and the server acknowledges the receipt of the data.
  - However, the server need not write the data to stable storage before sending the reply.
  - Asynchronous writes provide the server with several options to determine the best policy to synchronize the data.
  - The server may never schedule the write or may wait for multiple writes to be performed together

Later Versions of NFS

- NFS v3.0 introduced the asynchronous write operation.
- NFS v3.0 introduced basic file locking also via the Network Lock Manager.
  - This now becomes a stateful protocol.
- NFS v4.0 is a stateful protocol and adds support for
  - Compound operations: combine a sequence of operations to a single remote operation.
  - Other additional features: locking as a first class feature, leases.