

## CSE 451: Operating Systems Spring 2012

### Module 23 Distributed File Systems

Ed Lazowska  
lazowska@cs.washington.edu  
Allen Center 570

## Distributed File Systems

- A distributed file systems supports network-wide sharing of files and devices
- A DFS typically presents clients with a “traditional” file system view
  - there is a single file system namespace that all clients see
  - files can be shared
  - one client can observe the side-effects of other clients’ file system activities
  - in many (but not all) ways, an ideal distributed file system provides clients with the illusion of a shared, local file system
- But ...with a distributed implementation
  - read blocks / files from remote machines across a network, instead of from a local disk

© 2012 Gribble, Lazowska, Levy, Zahorjan

2

## DFS issues

- What is the basic abstraction
  - a remote file system?
    - open, close, read, write, ...
  - a remote disk?
    - read block, write block
- Naming
  - how are files named?
  - are those names **location transparent**?
    - is the file location visible to the user?
    - do the names change if the file moves?
    - do the names change if the user moves?

© 2012 Gribble, Lazowska, Levy, Zahorjan

3

- Caching
  - caching exists for performance reasons
  - where are file blocks cached?
    - on the file server?
    - on the client machine?
    - both?
- Sharing and coherency
  - what are the semantics of sharing?
  - what happens when a cached block/file is modified?
  - how does a node know when its cached blocks are stale?
    - if we cache on the client side, we’re presumably caching on multiple client machines if a file is being shared

© 2012 Gribble, Lazowska, Levy, Zahorjan

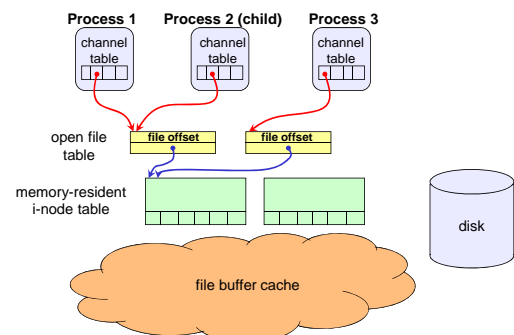
4

- Replication
  - replication can exist for performance and/or availability
  - can there be multiple copies of a file in the network?
  - if multiple copies, how are updates handled?
  - what if there’s a network partition? Can clients work on separate copies? If so, how does reconciliation take place?
- Performance
  - what is the performance of remote operations?
  - what is the additional cost of file sharing?
  - how does the system scale as the number of clients grows?
  - what are the performance bottlenecks: network, CPU, disks, protocols, data copying?

© 2012 Gribble, Lazowska, Levy, Zahorjan

5

## Reminder: Single-system Unix file sharing



© 2012 Gribble, Lazowska, Levy, Zahorjan

6

## Example: Sun's Network File System (NFS)

- The Sun Network File System (NFS) has become a common standard for distributed UNIX file access
- NFS runs over LANs (even over WANs – slowly)
- Basic idea
  - allow a remote directory to be “mounted” (spliced) onto a local directory
  - gives access to that remote directory and all its descendants as if they were part of the local hierarchy
- Pretty similar (except for implementation and performance) to a “local mount” or “link” on UNIX
  - I might link
 

```
/cse/www/education/courses/451/12sp/
```

 as
 

```
/u4/lazowska/451
```

 to allow easy access to my web data from my home directory:
 

```
cd
ln -s /cse/www/education/courses/451/12sp 451
```

© 2012 Gribble, Lazowska, Levy, Zahorjan

7

## NFS particulars

- ginger.cs* exports the directory *ginger.cs:/u4/lazowska*
- norton.cs* mounts this on */faculty/edl*
  - programs on *norton.cs* can access the remote directory *ginger.cs:/u4/lazowska* using the local path */faculty/edl*
- if, on *ginger.cs*, I had a file */u4/lazowska/myfile.txt*
  - programs on *norton.cs* could access it as */faculty/edl/myfile.txt*
- note that different clients might mount the same exported directory, but on different local paths
  - e.g., *forkbomb.cs* might mount it on */facultyfiles/edlazowska*
  - then, the file *ginger.cs:/u4/lazowska/myfile.txt* could be accessed with three different names
    - on *ginger.cs*: */u4/lazowska/myfile.txt*
    - on *norton.cs*: */faculty/edl/myfile.txt*
    - on *forkbomb.cs*: */facultyfiles/edlazowska/myfile.txt*

© 2012 Gribble, Lazowska, Levy, Zahorjan

8

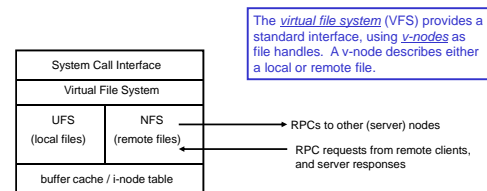
## NFS implementation

- NFS defines a set of RPC operations for remote file access:
  - searching a directory
  - reading directory entries
  - manipulating links and directories
  - reading/writing files
- Every node may be a client, a server, or both
  - E.g., a given machine might export some directories and import others

© 2012 Gribble, Lazowska, Levy, Zahorjan

9

- NFS defines new layers in the Unix file system



© 2012 Gribble, Lazowska, Levy, Zahorjan

10

## NFS caching / sharing

- On a file open, the client asks the server whether the client's cached blocks are up to date (good!)
  - but, once a file is open, different clients can perform concurrent reads and writes to it and get inconsistent data (bad!)
- Modified data is flushed back to the server every 30 seconds
  - the good news is this bounds the amount of inconsistency to a window of 30 seconds, and that this is simple to implement and understand
  - the bad news is that the inconsistency can be severe
    - e.g., data can be lost, different clients can see inconsistent states of the files at the same time

© 2012 Gribble, Lazowska, Levy, Zahorjan

11

## Example: CMU's Andrew File System (AFS)

- Developed at CMU to support all of its student computing
- Consists of workstation clients and dedicated file server machines (*differs from NFS*)
- Workstations have local disks, used to cache files being used locally (originally whole files, subsequently 64K file chunks) (*differs from NFS*)
- Andrew has a single name space – your files have the same names everywhere in the world (*differs from NFS*)
- Andrew is good for distant operation because of its local disk caching: after a slow startup, most accesses are to local disk

© 2012 Gribble, Lazowska, Levy, Zahorjan

12

## AFS caching/sharing

- Need for scaling required reduction of client-server message traffic
  - Once a file is cached, all operations are performed locally
  - On close, if the file has been modified, it is replaced on the server
- The client assumes that its cache is up to date, unless it receives a *callback* message from the server saying otherwise
  - on file open, if the client has received a callback on the file, it must fetch a new copy; otherwise it uses its locally-cached copy (*differs from NFS*)
- What if two users are accessing the same file?

© 2012 Gribble, Lazowska, Levy, Zahorjan

13



## Example: Berkeley Sprite File System

- Unix file system developed for *diskless* workstations with large memories (*differs from NFS, AFS*)
- Considers memory as a huge cache of disk blocks
  - memory is shared between file system and VM
- Files are permanently stored on servers
  - servers have a large memory that acts as a cache as well
- Several workstations can cache blocks for read-only files
- If a file is being written by more than 1 machine, client caching is turned off – all requests go to the server (*differs from NFS, AFS*)
  - So improved coherence, at higher cost

© 2012 Gribble, Lazowska, Levy, Zahorjan

14

## Example: Google's File System (GFS)

 <p>NFS, etc.</p>	 <p>GFS</p>
<p>Independence Small Scale Variety of workloads</p>	<p>Cooperation Large scale Very specific, well-understood workloads</p>

© 2012 Gribble, Lazowska, Levy, Zahorjan

15

## GFS: Environment

Why did Google build its own file system?

- Google has unique FS requirements
  - huge volume of data
  - huge read/write bandwidth
  - reliability over tens of thousands of nodes with frequent failures
  - mostly operating on large data blocks
  - needs efficient distributed operations
- Google has somewhat of an unfair advantage...it has control over, and customizes, its:
  - applications
  - libraries
  - operating system
  - networks
  - even its computers!

© 2012 Gribble, Lazowska, Levy, Zahorjan

16

## GFS: Files

- Files are huge by traditional standards (GB, TB, PB)
- Most files are mutated by appending new data rather than overwriting existing data
- Once written, the files are only read, and often only sequentially.
- Appending becomes the focus of performance optimization and atomicity guarantees
- **NOTE:** A major use of GFS is for storing event logs – what did you search for, which link did you follow, etc. Then these logs are mined for patterns. Hence huge, append-only, read sequentially.

© 2012 Gribble, Lazowska, Levy, Zahorjan

17

## GFS: Architecture

- A *GFS cluster* consists of a replicated *master* and multiple *chunk servers* and is accessed by multiple *clients*
- Each computer in the GFS cluster is typically a commodity Linux machine running a user-level server process
- Files are divided into fixed-size *chunks* identified by an immutable and globally unique 64-bit *chunk handle*
- For reliability, each chunk is *replicated* on multiple chunk servers
- The master maintains all file system metadata (like, on which chunk servers specific chunks are stored)
- The master periodically communicates with each chunk server in *HeartBeat* messages to determine its state
- Clients communicate with the master (to access metadata (e.g., to find the location of specific chunks)) and directly with chunk servers (to actually access the data)
  - Prevents the single master from becoming a bottleneck
- Neither clients nor chunk servers cache file data, eliminating cache coherence issues
  - Caching not helpful because most ops are huge sequential reads
- Clients do cache metadata, however
- If the master croaks, *Paxos* is used to select a new master from among the replicas

© 2012 Gribble, Lazowska, Levy, Zahorjan

18

