416 Distributed Systems

Distributed File Systems 2

Jan 22, 2018

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

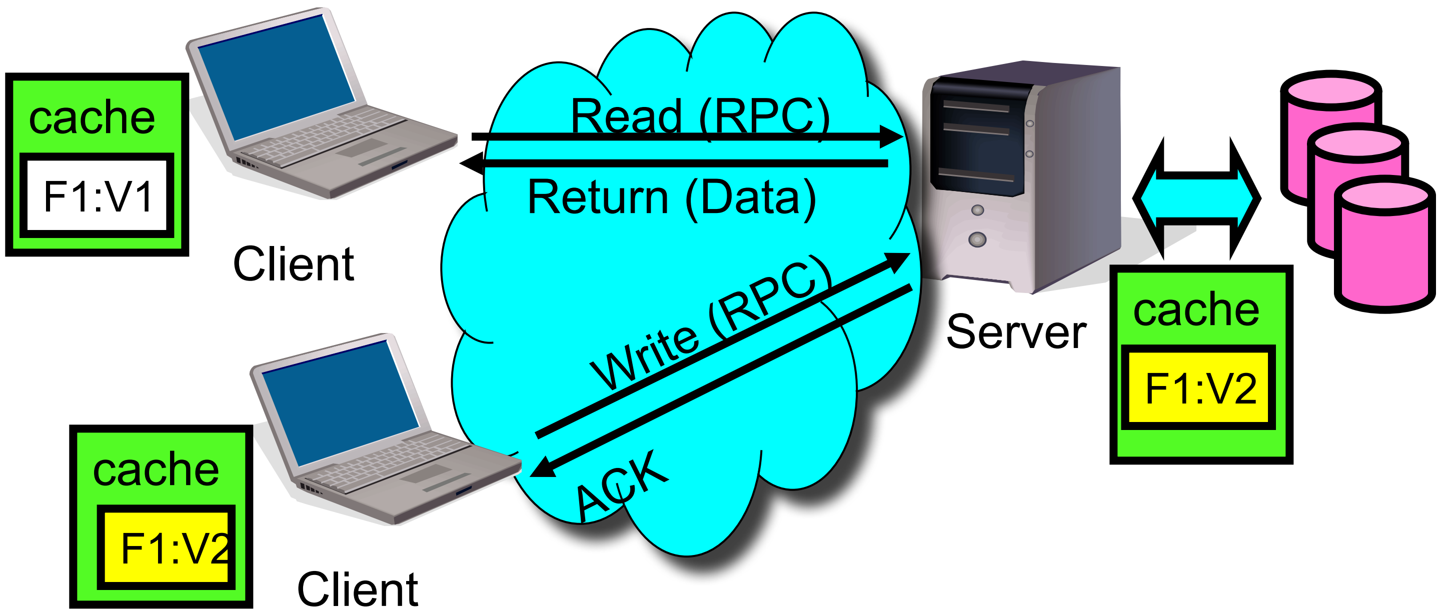
Topic 1: Client-Side Caching

- Many systems (not just distributed!) rely on two solutions to every problem:
  1. Cache it!
  2. *"All problems in computer science can be solved by adding another level of indirection. But that will usually create another problem."* -- David Wheeler

Use of caching to reduce network load (not AFS from assignment 2)

read(f1)→V1
read(f1)→V1
read(f1)→V1
read(f1)→V1

write(f1)→OK
read(f1)→V2



Client Caching in NFS v2

- Cache both clean and dirty file data and file attributes
  - Memory (e.g., DRAM) cache
- File attributes in the client cache expire after 60 seconds (file data doesn't expire)
- File data is checked against the modified-time in file attributes (which could be a cached copy)
  - Changes made on one machine can take up to 60 seconds to be reflected on another machine
- Dirty data are buffered on the client machine until file close or up to 30 seconds
  - If the machine crashes before then, the changes are lost

Implication of NFS v2 Client Caching

- Advantage: No network traffic if open/read/write/close can be done locally.
- But…. Data consistency guarantee is very poor
  - Simply unacceptable for some distributed applications
  - Imagine an application that modifies/reads a lot of shared state across multiple instances (e.g., distributed Game)
- Generally clients do not cache data on local disks

NFS' s Failure Handling – Stateless Server

- Files are state, but...
- Server exports files without creating extra state
  - No list of "who has this file open" (permission check on each operation on open file!)
  - No "pending transactions" across crash
- Crash recovery is "fast"
  - Reboot, let clients figure out what happened
- State stashed elsewhere
  - Separate MOUNT protocol
  - Separate NLM locking protocol
- Stateless protocol: requests specify exact state. read() → read([file], [position]). no seek on server.

NFS’ s Failure Handling

- Operations are idempotent
  - How can we ensure this?

NFS’s Failure Handling

- 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 (e.g., by same/other clients)

NFS's Failure Handling

- 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.
- Write-through caching: When file is closed, all modified blocks sent to server. close() does not return until bytes safely stored.
  - Close failures?
    - retry until things get through to the server
    - return failure to client
  - Most client apps can't handle failure of close() call.
  - Usual option: hang for a long time trying to contact server

NFS Results

- NFS provides transparent, remote file access
- Simple, portable, *really popular*
  - (it’s gotten a little more complex over time, but...)
- Weak consistency semantics
- Requires hefty server resources to scale (write-through, server queried for lots of operations)