File Access & Semantics of Sharing

- Overlapping / interleaving data access
  - When data is replicated in space to increase concurrency, *coherency control* is needed to keep the copies coherent
  - When data operations are spread out and interleaved in time, *concurrency control* is needed to prevent interference
  - Remote access — no local data
  - Cache access — small part kept locally
  - Dl/ul access — whole file is downloaded for local access, then uploaded

time / space	**remote access**	**cache access**	**down/up load access**
simple RW	no true sharing	coherency control	coherency control
transaction	concurrency control	concurrency control	concurrency control
session	not applicable	not applicable	ignore sharing

Semantics of Sharing (cont.)

- Grouping file operations in different time intervals:
  - Simple RW — each read & write operation is an independent request
  - Transaction — groups of reads and writes treated as an atomic action
  - Session — sequence of transactions and simple RW operations, plus additional semantics
- Three different semantic models:
  - UNIX semantics — result of a write goes immediately to the file, so reads always return the “latest” value
  - Transaction semantics — writes go to local storage and go to file when and if the transaction commits
  - Session semantics — similar, writes go to file when the session is closed

Cache Location

- No caching — all files on server’s disk
  - ✔ Simple, no local storage needed
  - ✘ Expensive transfers
- Cache files in server’s memory
  - ✔ Easy, transparent to clients
  - ✘ Still involves a network access
- Cache files on client’s local disk
  - ✔ Plenty of space, reliable
  - ✘ Faster than network, slower than memory
- Cache files in client’s memory
  - The usual solution (either in each process’s address space, or in the kernel)
  - ✔ Fast, permits diskless workstations
  - ✘ Data may be lost in a crash

Cache Modification Policy

- *Cache modification* (*writing*) *policy* decides when a modified (*dirty*) cache block should be *flushed* to the server
- *Write-through* — immediately flush the new value to server (& keep in cache)
  - ✔ No problems with consistency
  - ✔ Maximum reliability during crashes
  - ✘ Doesn’t take advantage of caching during writes (only during reads)
- *Write-back* (*delayed-write*) — flush the new value to server after some delay
  - ✔ Fast — write need only hit the cache before the process continues
  - ✔ Can reduce disk writes since the process may repeatedly write the same location
  - ✘ Unreliable — if machine crashes, unwritten data is lost

Cache Modification Policy (cont.)

- Variations on write-back (when are the new values flushed to the server?)
  - Write-on-close — flush new value to the server only when the file is closed
    - ✓ Can reduce disk writes, particularly when the file is open for a long time
    - ✗ Unreliable — if machine crashes, unwritten data is lost
    - ✗ May make the process wait on the file close
  - Write-periodically — flush new value to the server at periodic intervals (maybe 30 seconds)
    - ✓ Can only lose writes in last period

Cache Validation

- A client must decide whether or not a locally cached copy of data is consistent with the master copy
- *Client-initiated* validation:
  - Client initiates validity checks
  - Client contacts the server and asks if its copy is consistent with the server's copy
    - At every access, or
    - After a given interval, or
    - Only on file open
  - Server could enforce single-writer, multiple-reader semantics, but to do so
    - It would have to store client state (expensive)
    - Clients would have to specify access type (read / write) on open
  - ✗ High frequency of validity checks may mitigate the benefits of caching

Cache Validation (cont.)

- *Server-initiated* validation:
  - Server records the parts of each file that each client caches
  - Server detects potential conflicts if two or more clients cache the same file
  - Concurrency control for handling conflicts:
    - *Session semantics* — writes are only visible in sessions starting later (not to processes which have file open now)
      - When a client closes a file that it has modified, the server notifies the other clients that their cached copy is invalid, and they should discard it
        - » If another client has the file open, discard it when its session is over
    - *UNIX semantics* — writes are immediately visible to others
      - Clients specify the type of access they want when they open a file, so if two clients want to write the same file for writing, that file is not cached
  - ✗ Significant overhead at the server

Stateful vs. Stateless

- *Stateful server* — server maintains state information for each client for each file
  - Connection-oriented (open file, read / write file, close file)
  - ✔ Enables server optimizations like read-ahead (prefetching) and file locking
  - ✘ Difficult to recover state after a crash
- *Stateless server* — server does not maintain state information for each client
  - Each request is self-contained (file, position, access)
    - Connectionless (open and close are implied)
  - ✔ If server crashes, client can simply keep retransmitting requests until it recovers
  - ✘ No server optimizations like above
  - ✘ File operations must be idempotent

Caching in NFS

- Traditional UNIX
  - Caches file blocks, directories, and file attributes
  - Uses read-ahead (prefetching), and delayed-write (flushes every 30 seconds)
- NFS servers
  - Same as in UNIX, except server's write operations perform write-through
    - Otherwise, failure of server might result in undetected loss of data by clients
- NFS clients
  - Caches results of read, write, getattr, lookup, and readdir operations
  - Possible inconsistency problems
    - Writes by one client do not cause an immediate update of other clients' caches

Caching in NFS (cont.)

- NFS clients (cont.)
  - File reads
    - When a client caches one or more blocks from a file, it also caches a timestamp indicating the time when the file was last modified on the server
    - Whenever a file is opened, and the server is contacted to fetch a new block from the file, a validation check is performed
      - Client requests last modification time from server, and compares that time to its cached timestamp
      - If modification time is more recent, all cached blocks from that file are invalidated
      - Blocks are assumed to valid for next 3 seconds (30 seconds for directories)
  - File writes
    - When a cached page is modified, it is marked as dirty, and is flushed when the file is closed, or at the next periodic flush
  - Now two sources of inconsistency: delay after validation, delay until flush

Caching in Andrew

- When a remote file is accessed, the server sends the entire file to the client
  - The entire file is then stored in a disk cache on the client computer
    - Cache is big enough to store several hundred files
- Implements session semantics
  - Files are cached when opened
  - Modified files are flushed to the server when they are closed
  - Writes may not be immediately visible to other processes
- When client caches a file, server records that fact — it has a *callback* on the file
  - When a client modifies and closes a file, other clients lose their callback, and are notified by server that their copy is invalid

How Can Andrew Perform Well?

- Most file accesses are to files that are infrequently updated, or are accessed by only a single user, so the cached copy will remain valid for a long time
- Local cache can be big — maybe 100 MB — which is probably sufficient for one user's working set of files
- Typical UNIX workloads:
  - Files are small, most are less than 10kB
  - Read operations are 6 times more common than write operations
  - Sequential access is common, while random access is rare
  - Most files are read and written by only one user; if a file shared, usually only one user modifies it
  - Files are referenced in bursts