

The Sprite Distributed File System

- Based on “Caching in the Sprite Network File System”, by Nelson et al.
- This is another distributed file system case study (and will serve as a review, too)
- The traces of the Sprite study have been used repeatedly in the literature
- The paper is very readable

Overview

There are two main ideas in Sprite:

- Cache consistency (no stale data)
- Caches vary in size dynamically
 - this is not really a distributed file systems issue, but it is a good topic to cover

Sprite caches files in the main memory of clients and servers

- Clients only cache file data
- Servers can cache either data or metadata (e.g., directory information)
 - recall the distinction between real file service and directory service
- The end result is a consistent file system, but still not good enough to use for communication/synchronization

Motivation

- Sprite was motivated by a study of file system usage that has been widely used
- Findings of the study:
 - one third of all file accesses are writes
 - 75% of files are open less than 0.5 seconds
 - 90% of files are open less than 10 seconds
 - 20-30% of new data is deleted within 30 seconds
 - 50% is deleted within 5 minutes
 - file sharing is rare
- What do these numbers mean?
 - think in terms of caching mechanism...

Write Policy in Sprite

- Recall that we want delayed writing for performance
 - minor reliability problems are generally ignored
- Several file systems employ “write-on-close” instead of “write-through”.
- Problem: it may still be too soon
 - 75% of files are open less than 0.5 secs
 - 90% of files are open less than 10 secs
- Sprite does periodic writes instead
 - every 30 seconds blocks that have not been modified for 30 seconds are written back
 - a block will go to the server’s cache in 30-60 secs and to disk in 60-120 secs
- Problem: this may be too infrequent if files are used for communication/synchronization

Cache Consistency

- Let's distinguish between “sequential write sharing” and “concurrent write-sharing”
 - *sequential write-sharing*: the file is shared but is opened for reading and writing in turn, not simultaneously
 - *concurrent write sharing*: readers and writers can open the file concurrently
- Most distributed file systems guarantee strong consistency for sequential write sharing
 - Even NFS:
 - NFS write policy is “write-on-close”
 - when a file is opened, the version of cached blocks is checked against the server version
- Sprite also guarantees strongly consistent concurrent write sharing

Sprite Cache Consistency

- Main idea:
 - all file requests go through a server
 - the server keeps state about the clients, so that it knows when a file is about to be concurrent-write shared
 - in this case caching is disabled on all clients
- Mechanism for disabling caching:
 - the server notifies all clients in case of concurrent write sharing
 - the write client has to write all changed blocks back to the server
 - read clients have to discard their cached blocks and direct all future read requests to the server
- Clearly this is not too efficient, but concurrent write sharing is supposed to be rare
 - not so if files are used for communication/synchronization

Sprite Cache Consistency (cont'd)

- Recall that Sprite delays writes for fixed amounts of time
 - instead of doing “write-on-close”
- Thus, the NFS policy is not enough to prevent *sequential* write sharing inconsistencies
- Instead: when a file is opened, the last writer is notified so that it can write back the changed blocks

Dynamically Varying Cache Size

- RAM is used for caching both file blocks and virtual memory pages
 - two separate caches in most systems
- Sprite uses an approximate LRU algorithm for dynamically sizing the two caches
 - the time of last access is kept for every block
 - the “oldest” block is replaced
 - for virtual memory “oldest” is approximate (common optimization)
- can be complicated by different page/block sizes

- For swap files, the file block cache is bypassed
 - i.e., no delayed write-back
 - but pages *will* be cached in server RAM (and access faster than on local disk maybe!)
- Executable (i.e., read-only) file pages need to be mapped to VM, but they are already in file cache (e.g., just compiled)
 - the pages move from one cache to the other, get marked for replacement in file cache

Performance Measurements

- client caches (with delayed write) important for
 - client performance
 - server utilization
 - network utilization
 - scalability
- caching directory/naming/attributes -> further reduces server/network utilization; need more state at server

Other issues

- recovery
- disk overflow