Some Topics in Distributed Operating Systems

Distributed Operating System

- Operating system that you have studied so far manages resources in a single machine
  - CPUs, memory, file systems, disk, …
- Distributed Operating Systems
  - manages resources spread over multiple machines interconnected by some network, each machine running its own operating system
  - Attempts to make the set of machines (nodes) appear as a single node to the user

- Functionalities of a distributed operating system (drawing from analogy with a single machine OS)
  - Scheduling a task in one of the machines
    - Where can a task be submitted?
      - One fixed machine, fixed subset of machine, any machine, …
    - Who decides which machine the task is to be run?
      - Centralized vs. distributed
    - How to decide which machine the task is to be run?
      - Resource matching, load balancing, …

- Synchronization issues
  - Enforcing ordering of execution of tasks in different machines
  - Distributed mutual exclusion
  - Distributed deadlock detection and handling
- Memory management
  - Distributed shared memory or not
- Distributed file systems
  - Files spread over multiple machines
- Security
  - Distributed authentication, access control, …

- There are other functionalities that are unique to a distributed system
  - Clock synchronization and ordering of events, Global snapshot, Leader election, …
  - In any distributed system, failure of components must be considered
    - Any mechanism used should be fault tolerant
- Given the very short time we have, we will just look briefly at a few topics
  - Mostly without failures

Distributed Mutual Exclusion

Mutual Exclusion

- Well-understood in shared memory systems
- Requirements:
  - at most one process in critical section (safety)
  - if more than one requesting process, someone enters (liveness)
  - a requesting process enters within a finite time (no starvation)
  - requests are granted in some order (fairness)

Some Complexity Measures

- No. of messages per critical section entry
- Synchronization delay
- Response time
- Throughput

Classification of Distributed Mutual Exclusion Algorithms

- **Permission based**
  - Node takes permission from all/subset of other nodes before entering critical section
  - Permission from all: costly, good for small systems
  - Permission from subset: scalable, widely used
    - Main problem: How to choose the subsets for each node?
- **Token based**
  - Single token in the system
  - Node enters critical section if it has the token
  - Algorithms differ in how the token is circulated among requesting nodes

Adaptive vs. Non-Adaptive

- Performance under low load (less requests for CS) should be better than performance under high load (lots of requests)
- **Adaptive** mutual exclusion algorithms: performance is dependent on load

Permission based Algorithms: Example

Maekawa's Algorithm

- Permission obtained from only a subset of other processes, called the Request Set (or Quorum)
- Separate Request Set R_i for each process i
- Requirements:
  - for all i, j: $R_i \cap R_j \neq \Phi$
  - for all i: $i \in R_i$
  - for all i: $|R_i| = K$, for some K
  - any node i is contained in exactly D Request Sets, for some D
- $K = D$ (easy to see)
- For minimum K, $K \approx \text{sqrt}(N)$ (why?)

A simple version

- To request critical section:
  - i sends REQUEST message to all processes in R_i
- On receiving a REQUEST message:
  - send a REPLY message if no REPLY message has been sent since the last RELEASE message is received. Update status to indicate that a REPLY has been sent. Otherwise, queue up the REQUEST
- To enter critical section:
  - i enters critical section after receiving REPLY from all nodes in R_i

- To release critical section:
  - send RELEASE message to all nodes in R_i
  - On receiving a RELEASE message, send REPLY to next node in queue and delete the node from the queue. If queue is empty, update status to indicate no REPLY message has been sent since last RELEASE is received.

- Message Complexity: 3*sqrt(N)
- Synchronization delay =

  2 *(max message transmission time)
- Major problem: Deadlock possible
- Can you update the protocol with additional messages to solve this problem?
  - Good practice ☺
  - Maekawa's protocol already does that, we just looked at a part of it
- Building the request sets?

Some Observations

- Permission based algorithms with permission from a subset are widely used
  - Voting/Quorum based protocols
    - In Maekawa's algorithm,
      - each process has one vote
      - A process needs a certain number of votes to proceed
- Questions/Issues
  - How to choose the quorums?
  - Should the quorum be the same for read and write?
  - Should each process have one vote only? Same number of votes for all?
  - Dynamic quorums/votes

Token based Algorithms: Example

Token based Algorithms

- Single token circulates, enter CS when token is present
- Mutual exclusion obvious
- Algorithms differ in how to find and get the token
  - Token circulates, nodes use it when it passes through them
  - Token stays at node of last use, other nodes request for it when needed
    - Need to differentiate between old and current requests

Suzuki Kasami Algorithm

- Broadcast a request for the token
- Process with the token sends it to the requestor if it does not need it

Issues:

- Current vs. outdated requests
- Determining sites with pending requests
- Deciding which site to give the token to

- The token:
  - Queue (FIFO) Q of requesting processes
  - LN[1..n] : sequence number of request that j executed most recently
- The request message:
  - REQUEST(i, k): request message from node i for its k^{th} critical section execution
- Other data structures
  - $RN_i[1..n]$ for each node i, where $RN_i[j]$ is the largest sequence number received so far by i in a REQUEST message from j.

- To request critical section:
  - If i does not have token, increment $RN_i[i]$ and send REQUEST(i, $RN_i[i]$) to all nodes
  - if i has token already, enter critical section if the token is idle (no pending requests in token Q), else follow rule to release critical section

- On receiving REQUEST(i, sn) at j:
  - set $RN_j[i] = \max(RN_j[i], sn)$
  - if j has the token and the token is idle, send it to i if $RN_j[i] = LN[i] + 1$. If token is not idle, follow rule to release critical section

- To enter critical section:
  - enter CS if token received after sending REQUEST

- To release critical section:
  - set LN[i] = RNi[i]
  - For every node j which is not in Q (in token), add node j to Q if RNi[j] = LN[j] + 1
  - If Q is non empty after the above, delete first node from Q and send the token to that node

Points to note:

- No. of messages: 0 if node holds the token already and token is idle, n otherwise
- Synchronization delay: 0 (node has the token) or max. message delay (token is elsewhere)
- No starvation

Distributed File Systems

Distributed File Systems

- DFS : distributed implementation of a file system
  - Files, file servers, and users are all dispersed around the network
- Main goal: DFS should look like a centralized file system to an user
  - Ability to open & update any file on any machine on network
  - Ability to share files same as shared local files

Design Goals

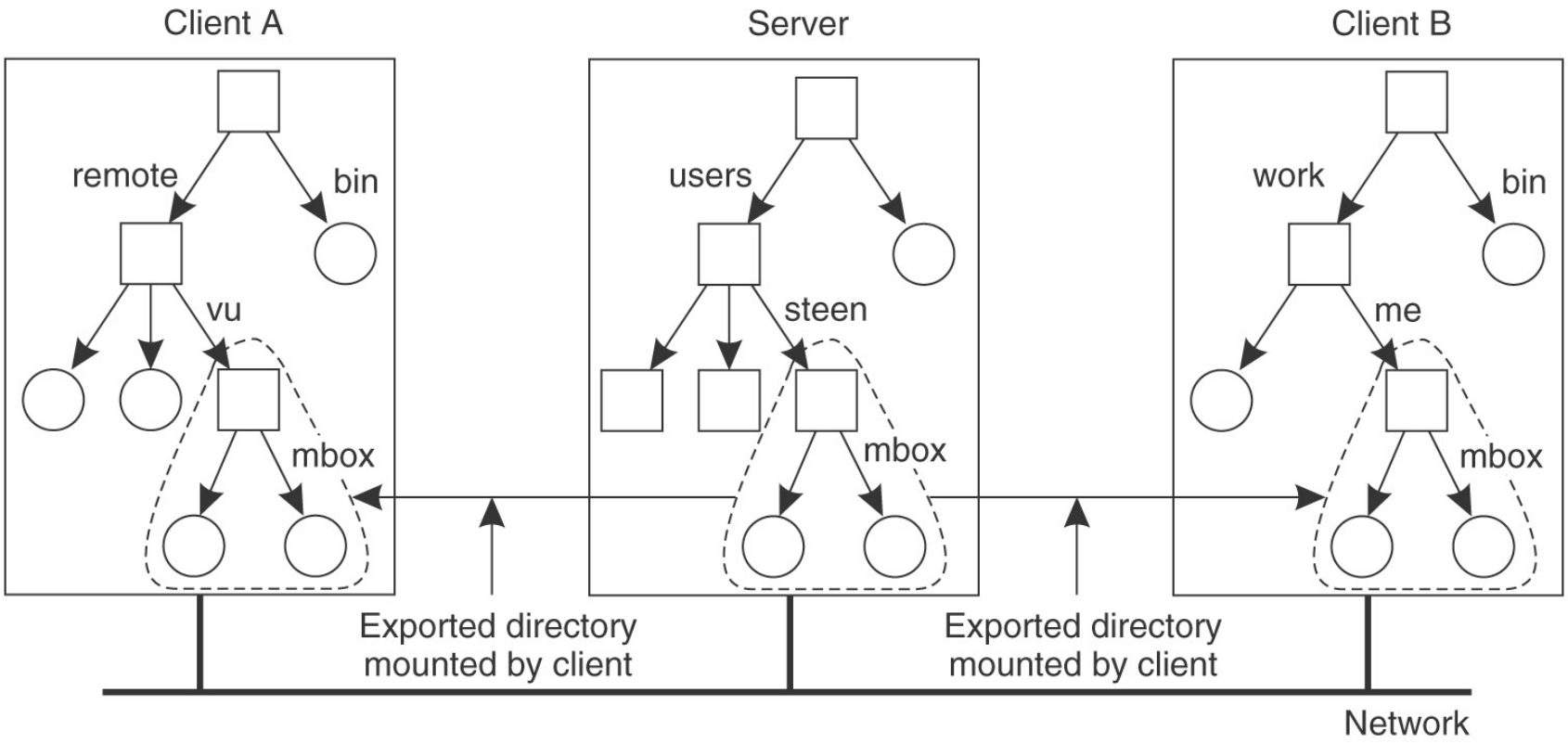
- Network transparency
  - Same set of file operations for both local and remote files
  - Uniform naming scheme for local and remote files
  - Similar access time for local and remote files
- High availability
  - Files should not become unavailable for "small" failures
- Scalability
  - no. of users, file servers, files handled etc.
- Concurrency
  - Handle concurrent access by different clients in the network

Design Issues: Naming Schemes

- Files named by combination of host name and local name
  - Not location transparent
- Single global namespace spanning all files
  - Location transparent
  - All systems see the same directory structure
  - A server crash affects all machines
  - Example: Andrew File System (AFS)

- Mount remote directories to local directories
  - Location transparent
  - Directory structure seen by different machines possibly different
  - Mount only when needed, what is needed
  - Mounted remote directories can be accessed transparently
  - Example: Network File System (NFS)

Mounting Remote Directories (NFS)



Pathname Translation

- Consider /remote/vu/mbox in A
- /remote/vu looked up in A the usual way
- Low level entry for vu contains information that it is actually /users/steen in server
- Request sent to server to look up /users/steen/mbox
- What if the name crosses multiple machine boundaries?

Caching to Improve Performance

- Reduce network traffic by retaining recently accessed disk blocks in local cache
  - Repeated accesses to the same information can be handled locally from the cached copy
- If data accessed not in cache, copy of data brought from the server to the local cache
  - Copies of parts of file may be scattered in different caches
- Cache-consistency problem – keeping the cached copies consistent with the master file in the presence of write operations

Design Issue: Caching

- Cache location
- Granularity of cached data
- Update policy
- Cache consistency

Cache Location

- In client memory
  - Faster access
  - Good when local usage is transient
  - Enables diskless workstations
- On client disk
  - Good when local usage dominates
  - Can cache larger files
  - Helps protect clients from server crashes

Granularity of Cached Data

- Whole file
  - Easier to maintain consistency
  - High transfer time, specially if only a small part of the file is accessed
- One or more blocks
  - Low transfer time, transfer as and when needed
  - More complex consistency issues
- In general, increasing granularity means higher chance of finding next accessed data, but also higher transfer time

Cache Update Policies

- When is cached data at the client updated in the master file?
- Write-through – write data through to disk when cache is updated
  - Reliable, but poor performance
- Delayed-write – cache and then write to the server later
  - Write operations complete quickly; some data may be overwritten in cache, saving needless network I/O
  - Poor reliability
    - Unwritten data may be lost when client machine crashes
    - Inconsistent data
  - Variation – write dirty blocks back at regular intervals

Cache Consistency

- Is locally cached copy of the data consistent with the master copy?
- Client-initiated approach
  - Client initiates a validity check with server
  - Server verifies local data with the master copy
    - Can use timestamps, version no….
- Server-initiated approach
  - Server records (parts of) files cached in each client
  - When server detects a potential inconsistency, it invalidates the client caches

Sharing Semantics in DFS

- Unix Semantics
  - Read after write
  - Every change becomes instantly visible to all processes
- Session Semantics
  - Same as unix semantics for local clients of a file
  - For remote clients, changes to a file are initially visible to the process that modified the file. Only when the file is closed, the changes are visible to other processes.
- Immutable files

Stateless vs Stateful Service

- Stateless service
  - No open and close calls to access files
  - Each request identifies the file and position in file
  - No client state information in server by making each request self-contained
  - Advantages
    - Better failure recovery
  - Disadvantage
    - Longer request messages, longer processing time

- Stateful Service
  - Client opens a file
  - Server fetches information about file from disk, stores in server memory
    - Returns to client a connection identifier unique to client and open file
    - Identifier used for subsequent accesses until session ends
  - Server must reclaim space used by no longer active clients
  - Advantages
    - Increased performance; fewer disk accesses
  - Disadvantages
    - Poor failure recovery

Network File System (NFS)

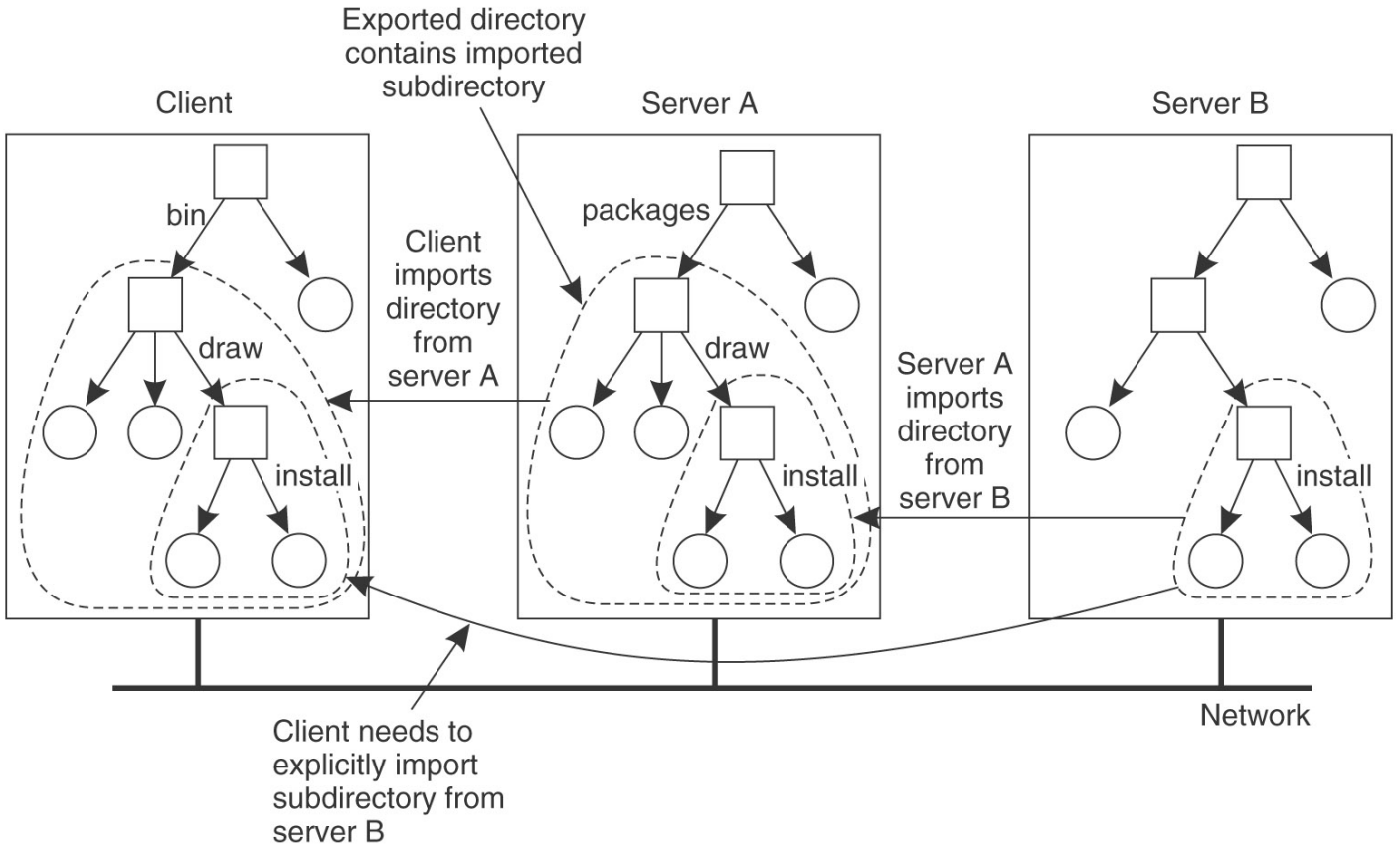
NFS

- Introduced by Sun Microsystems in 1984
- v2 in 1989, v3 in 1995, v4 in 2000 (updated in 2003)
- de facto standard for distributed file access
  - NFSv3 still widely used
  - Current OSs support both v3 and v4
- NFS normally runs over LAN
  - Can run on WAN also, though slower
- Any system may be both a client and server

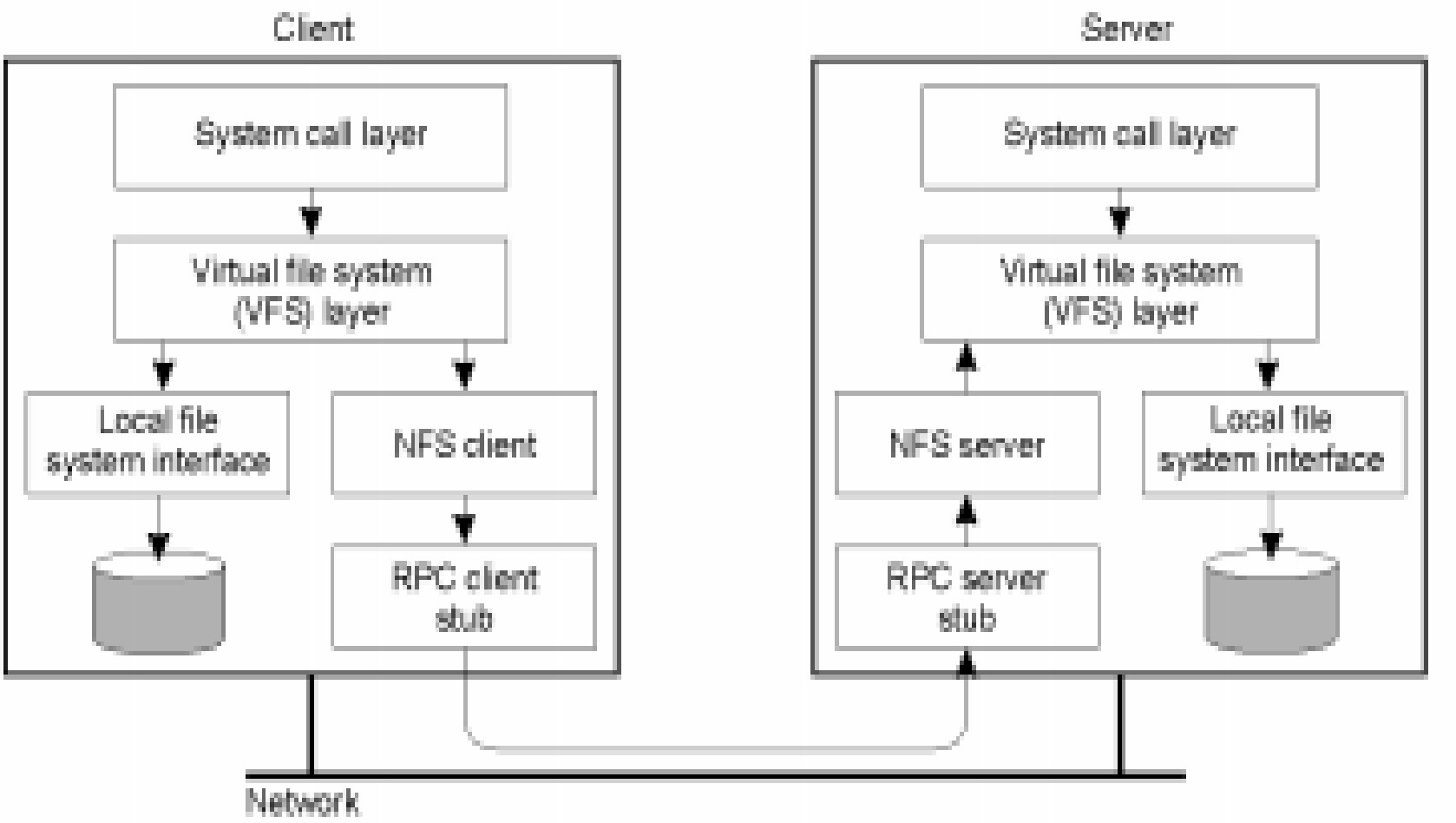
Main Idea

- Server exports parts of a filesystem to clients (Ex. specified in /etc/exports file in Linux)
- Clients "mount" server exported namespace at specific mount points in its namespace tree (specified in /etc/fstab in Linux; can be mounted separately by mount command also)
- Same server filesystem can be mounted at different mount points at different clients, so no single global name space
- Mounts can be cascaded
  - But client must mount from different servers explicitly itself

Nested Mounting (NFS v3)



Basic NFS Architecture



- Location transparent naming (host id needed only during mount)
- Pathname translation
  - Look up each component in the pathname recursively
  - Lookup call made to server when a mount point is crossed
  - Cascading mounts can involve multiple servers during translation

- Set of operations provided for operations on files
  - Lookup, read, write, mkdir, rename, rmdir, readdir, getattr, setattr…
  - No open or close calls (stateless server)
- Most data-modifying calls are synchronous
- Most calls work on an opaque (to client) file handle returned to the client by the server
  - Uniquely identifies the file in the server
- RPC (Remote Procedure Call) used to communicate between server and client

Caching in NFSv3

- Strictly not part of NFS protocol, but used in practice for performance
  - File attributes cached along with file block
  - Cached file block used subject to consistency checks on file attributes
  - 8 Kb blocks for v2, negotiable for v3
- No locking, no synchronization as part of NFS
  - Separate centralized locking mechanism using lockd
- No clear semantics because of caching nature

Stateless Service

- Server keeps no information of client
- Every client operation provides file handle
- Server caching for performance only
  - Based on recent usage (read ahead block)
- Client caching
  - Client checks validity of cached files (using time stamps)
  - Client responsible for writing out caches
- A cache entry is valid if one of the following is true
  - Cache entry is less than t seconds old
  - Last Modified time of file at server is the same as last modified time of file on client