Replication

- Pros: reliability and performance
  - Data is replicated to increase the reliability of the system. Tolerate site failures and network partition.
  - Replicating the server can reduce the server workload, and make the system scalable, especially in a large geographical area.
- Cons: price of making replicas consistent.
  - For example, clients can use web cache to improve performance. How to deal with web server updates?
  - One solution: no cache, low performance.
  - Another solution: the server invalidates or updates each cached copy. It needs to keep track of all caches, and sending them messages, and degrade the performance.

Replication and Caching

- Replication and caching are widely used for scalability. However, consistency issues against replication and caching.
  - Not good when the access-to-update ratio is very low. Keeping the replica or cache consistent may generate more network traffic and affect the scalability.
- Intuitively, replicas are consistent when the replicas are always the same. Thus, the update should be propagated to all replicas.
- Difficult to implement and not necessary in most cases. A need to relax consistency models.

Assumptions
- average object size = 1000,000 bits
- avg. request rate from institution’s browsers to origin servers = 15/sec
- delay from institutional router to any origin server and back to router = 2 sec

Consequences
- utilization on LAN = 15%
- utilization on access link = 100%
- total delay = Internet delay + access delay + LAN delay = 2 sec + minutes + milliseconds

possible solution: increase bandwidth of access link to, say, 100 Mbps
Consequence: utilization on LAN = 15%, utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay = 2 sec + msecs + msecs
- often a costly upgrade
With Caching

possible solution: install cache
• suppose hit rate is 0.4
consequence
• 40% requests will be satisfied almost immediately
• 60% requests satisfied by origin server
• utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)
• total avg delay = Internet delay + access delay + LAN delay = .6*(2.01) secs + .4*milliseconds < 1.4 secs

Consistency Models

• Maintaining perfect consistency is especially painful when the various copies are on different machines that can only communicate by sending messages over a slow network.
• In some applications, the solution is to accept less than perfect consistency as the price for better performance.
• For example, in distributed shared memory, a consistency model is essentially a contract between the software and the memory. It says that if the software agrees to obey certain rules, the memory promises to work correctly.

Generalized Consistency Models

• Consistency model has been studied in distributed shared memory, distributed shared database, and distributed file system. Here, we use the broader term data store.
• A data store may be physically distributed across multiple machines. Read or write operation.
• A consistency model is essentially a contract between processes and the data store.

Continuous Consistency

• There is no best solution to replicating data. There is no general ways to lose consistency: exactly what can be tolerated depends on applications.
• Continuous consistency provides three axes for defining inconsistency:
  – deviation in numerical values between replicas, e.g., stock price value difference of $0.01.
  – deviation in staleness between replicas,
  – deviation with respect to the ordering of update operations.
• Consistency unit (conit) specifies the unit over which consistency is to be measured.
Both x, y are initialized to 0. Vector clock \( <t,i> \) denotes operation that was carried out by replica i at its logical time t.

Replica A has three tentative update operations, which brings its ordering deviation to 3.

The only operation A has not yet seen is \(<10,B>\), bring its numerical deviation to 1. The weight of this deviation is the difference between values of x, y at A and the result at B not seen by A, which is 5.

There is a tradeoff between maintaining fine-grained and coarse-grained conits.
- If a conit represents more data, the replica may be in an inconsistent state quickly.
- Making conits very small may increase the total number of conits to be managed.

Continuous consistency can be implemented as a toolkit

\[
\text{DependsOnConit(ConitQ, 4, 0, 60);} \\
\text{// specifies the numerical, ordering, and staleness deviation}
\]

Read message m from head of queue Q;

consistent ordering of operations: strict consistency

Besides continuous consistency, there are many other consistency models. When tentative updates at replicas need to be committed, replicas will need to reach agreement on a global ordering of those updates.

In strict consistency, any read to a data item x returns the value stored by the most recent write operation to x.

Not practical when considering the network propagation delay.
Sequential Consistency

- The result of an execution is the same as if the operations of all processors are executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.
- Any valid interleaving of read/write is acceptable, but all processes must see the same interleaving of operations.

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
<td>z = 1;</td>
</tr>
<tr>
<td>print(y,z);</td>
<td>print(x,z);</td>
<td>print(x,y);</td>
</tr>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
<td>z = 1;</td>
</tr>
<tr>
<td>print(y,z);</td>
<td>print(x,z);</td>
<td>print(x,y);</td>
</tr>
<tr>
<td>y = 1;</td>
<td>print(x,z);</td>
<td>print(x,y);</td>
</tr>
<tr>
<td>z = 1;</td>
<td>print(y,z);</td>
<td>print(y,z);</td>
</tr>
<tr>
<td>print(x,y);</td>
<td>print(x,y);</td>
<td>print(x,z);</td>
</tr>
</tbody>
</table>

Prints: 001011  Prints: 101011  Prints: 010111  Prints: 111111

- Sequential consistency is similar to serializability of transactions. The difference is granularity: sequential consistency is in terms of read and write, whereas serializability is in terms of transactions.
- Many sequences are impossible, for example, 000000

Linearizability

- Linearizability states that: Each operation should appear to take effect instantaneously at some moment between its start and completion.

<table>
<thead>
<tr>
<th>Ordering of operations</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₁(x)a; W₂(y)b; W₁(y)a; W₂(x)b</td>
<td>R₁(x)b; R₂(y)b</td>
</tr>
<tr>
<td>W₁(x)a; W₂(y)b; W₁(y)a; W₂(x)b</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
<tr>
<td>W₁(x)a; W₂(y)b; W₂(x)b; W₁(y)a</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
<tr>
<td>W₂(y)b; W₁(x)a; W₂(x)b; W₁(y)a</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
<tr>
<td>W₂(y)b; W₁(x)a; W₂(x)b; W₁(y)a</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
<tr>
<td>W₂(y)b; W₂(x)b; W₁(x)a; W₁(y)a</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
</tbody>
</table>

- R₁(x)a R₂(y)b is impossible

<table>
<thead>
<tr>
<th>Ordering of operations</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₁(x)a; W₂(y)b; W₁(y)a; W₂(x)b</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
<tr>
<td>W₁(x)a; W₂(y)b; W₂(x)b; W₁(y)a</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
<tr>
<td>W₂(y)b; W₁(x)a; W₁(y)a; W₂(x)b</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
<tr>
<td>W₂(y)b; W₁(x)a; W₂(x)b; W₁(y)a</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
<tr>
<td>W₂(y)b; W₂(x)b; W₁(x)a; W₁(y)a</td>
<td>R₁(x)b; R₂(y)a</td>
</tr>
</tbody>
</table>

- Linearizable sequential consistency has more constraints, but it also eases programmability.
Causal Consistency

- Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

\[
\begin{align*}
P_1: & \quad W(x)a \quad W(x)c \\
P_2: & \quad R(x)a \quad W(x)b \\
P_3: & \quad R(x)a \quad R(x)c \quad R(x)b \\
P_4: & \quad R(x)a \quad R(x)b \quad R(x)c
\end{align*}
\]

Sequential and causal consistency are defined at the level of read/write.

- The application may need coarse granularity, by using concept similar to monitor in mutual exclusion.

- Necessary criteria for correct synchronization (entry consistency):
  - An acquire access of a synchronization variable, not allowed to perform until all updates to guarded shared data have been performed with respect to that process.
  - Before exclusive mode access to synchronization variable by process is allowed to perform with respect to that process, no other process may hold synchronization variable, not even in nonexclusive mode.
  - After exclusive mode access to synchronization variable has been performed, any other process’ next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable’s owner.

\[
\begin{align*}
P_1: & \quad Acq(Lx) \quad W(x)a \quad Acq(Ly) \quad W(y)b \quad Rel(Lx) \quad Rel(Ly) \\
P_2: & \quad Acq(Lx) \quad R(x)a \quad R(y) \quad NIL \\
P_3: & \quad Acq(Ly) \quad R(y)b
\end{align*}
\]

Eventual Consistency

- Sequential, causal, entry consistency prove a system wide consistent view on a data store.

- In a special class of distributed data stores, most operations are read. There are no simultaneous updates, or concurrent updates can be easily resolved.
  - Many examples: DNS, only the authoritative DNS server can update. In Web, only the web server updates. Cache can be used.

- Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. It is also called optimistic replication, is widely deployed in distributed systems, and has origins in early mobile computing projects.

- Consider a calendar shared between Alice, Bob, and Chuck. When Alice proposes to start meeting M at time T, and assuming no one else has confirmed attendance, she executes the operation \(W_A(M)[T, \{A\}]\). When Bob confirms his attendance, he will have read the tuple \([T, \{A\}]\) and update M accordingly:

\[
\begin{align*}
W_A(M_1)[T_1, \{A\}] & \rightarrow R_B(M_1)[T_1, \{A\}] \\
W_B(M_1)[T_1, \{A, B\}] & \rightarrow W_B(M_2)[T_2, \{B\}], \\
W_B(M_1)[T_1, \{A, B\}] & \quad || \quad W_C(M_1)[T_3, \{C\}]
\end{align*}
\]

- Three possible outcomes: \([T_1, \{A\}] \quad [T_1, \{A, B\}] \quad [T_3, \{C\}]

- With causal consistency, once a process reads \([T_2, \{B\}]\), \([T_1, \{A\}]\) is impossible, i.e.; either \([T_1, \{A, B\}]\) or \([T_3, \{C\}]\)
Consistency for Mobile Users

- Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.
  - At location A you access the database doing reads and updates.
  - At location B you continue your work, but unless you access the same server as the one at location A, you may detect inconsistencies:
    - your updates at A may not have yet been propagated to B
    - you may be reading newer entries than the ones available at A
    - your updates at B may eventually conflict with those at A

- The only thing you really want is that the entries you updated and/or read at A, are in B the way you left them in A. In that case, the database will appear to be consistent to you.

Client-centric Consistency

- The problem of operating on different replica can be solved by introducing client-centric consistency, which provides guarantees for a single client concerning the consistency of accesses to a data store by that client. No guarantees are given concerning different clients.
- The solution is based on Bayou, which is a Database system developed for mobile computing.
- Bayou distinguishes four consistency models: monotonic reads, monotonic writes, read your writes, writes follow reads.
- All read/write are performed on local copy. Updates are eventually propagated to other copies. Each data item has an associated owner, so no write-write conflicts.

Monotonic Reads

- If a process reads the value of a data item \( x \), any successive read operation on \( x \) by that process will always return that same value or a more recent value.

- \( W_1(x_2) \) is the write operation by process \( P_1 \) that leads to version \( x_2 \) of \( x \).
- \( W_1(x_i; x_j) \) indicates \( P_1 \) produces version \( x_j \) based on a previous version \( x_i \).
- \( W_1(x_j|x_i) \) indicates \( P_1 \) produces version \( x_j \) concurrently to version \( x_i \).
**Examples**

- Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.
- Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

**Monotonic Writes**

- A write operation by a process on a data item \( x \) is completed before any successive write operation on \( x \) by the same process.

\[
\begin{array}{c}
L_1: W_i(x_j) \\
L_2: W_j(x_i ; x_j) \quad W_i(x_j ; x_i)
\end{array}
\]

- Updating a program at server S2, and ensuring that all components on which compilation and linking depends, are also placed at S2.
- Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

**Read Your Writes**

- The effect of a write operation by a process on data item \( x \) will always be seen by a successive read operation on \( x \) by the same process.

\[
\begin{array}{c}
L_1: W_i(x_j) \\
L_2: W_j(x_i ; x_j) \quad R_i(x_j)
\end{array}
\]

- Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.

**Writes Follow Reads**

- A write operation by a process on a data item \( x \) following a previous read operation on \( x \) by the same process is guaranteed to take place on the same or a more recent value of \( x \) that was read.

\[
\begin{array}{c}
L_1: W_i(x_j) \quad R_i(x_j) \\
L_2: W_j(x_i ; x_j) \quad W_i(x_j ; x_i)
\end{array}
\]

- Example: see reactions to posted articles only if you have the original posting (a read "pulls in" the corresponding write operation).
Replica Placement

- Where, when and by whom replicas should be placed?
  Replica placement has two sub-problems:
  - Placing replica servers: the location to place a server
  - Placing content: finding the best servers for placing the content.
- Replica-server placement
  - Optimization problem in which the best \( K \) out of \( N (K<N) \) locations to be selected. Computationally complex, and solve with heuristics.
  - Selects one server at a time such that the average distance between that server and its clients is minimal given that \( k \) servers have been placed.
  - Another solution is to ignore the position of clients and only take the topology of the Internet as formed by the autonomous systems (AS).
  - First consider the largest AS and place a server on the router with the largest number of links. Then the second largest. ..

Facility location problem

The facility location problem is defined as follows. Given a set of locations \( i \) at which facilities may be built, building a facility at location \( i \) incurs a cost of \( f_i \). Each client \( j \) must be assigned to one facility, incurring a cost of \( d_j c_{ij} \) where \( d_j \) denotes the demand of the node \( j \), and \( c_{ij} \) denotes the distance between \( i \) and \( j \). The objective is to find a solution (i.e., both the number of facilities and the locations of the facilities) of the minimum total cost.

Fast Replica-Server Placement

- Using region, which is a collection of nodes accessing the same content, but for which the internode latency is low.
- A node in the most demanding region will become a replica server.
- The cell size is a function of the distance between two nodes and the number of required replicas.

Content Replication and Placement

- Permanent replicas
  - A number of web servers. A scheduler decides which replica to use
  - Mirror sites, where the client chooses a site to get the data.
- Server-initiated replicas
  - Dynamically created at the initiative of the server (data store) due to load increases.
  - Web hosting companies need to decide where and when replicas should be created and deleted.
  - Two issues: replication can take place to reduce the load on the server; specific files on a server can be migrated or replicated to servers close to the clients.
- Client-initiated replicas
  - At the request of the client, also called client cache
• Counting access requests from different clients.
• When the number of requests for $F$ at $S$ drops below a deletion threshold $\text{del}(S, F)$, that file can be removed from $S$. Make sure that at least one copy exists
• To add replica based on replication threshold $\text{rep}(S, F)$.

Server-Initiated Replicas

• When # of requests is between $\text{rep}(S, F)$ and $\text{del}(S, F)$, the file is allowed only to be migrated.
• If for some server $P$, $\text{cnt}_Q(P, F)$ exceeds half of the total requests for $F$ at $Q$, $Q$ attempts to migrate $F$ to $P$.
  – Migration may not be successful due to overload or lack of space. In this case, $Q$ tries to replicate $F$ on other servers.
  • # of requests is larger than $\text{rep}(Q, F)$
  • Starting from the farthest server $(R)$, if $\text{cnt}_Q(R, F)$ exceeds a certain fraction of all requests for $F$ at $Q$, replicate $F$ to $R$.
• Server-initiated replica becomes popular due to web hosting. Permanent replicas are still useful and can be used to guarantee consistency whereas server-initiated replicas are used for placing read-only copies close to clients.

Client-Initiated Replicas

• Client side cache can improve performance.
• After a read, keep the data in local cache.
• To improve the cache hits, share cache with other clients.
  – This may not be very popular in file system, but popular in web
• Client cache can be in local machine, or in the local network, or even shared within department or organization.

Content Distribution (Update Propagation)

• Updates are generally initiated at a client and then forwarded to other replicas. Design issues to consider:
  • State vs. operations
  • Pull vs push
  • Multicast vs unicast
  • State vs. operations
    • Propagate only a notification of an update (invalidation)
      • good when the read-to-write ratio is relatively small since several writes may overwrite each other.
    • Transfer data from one to another
      • good when read-to-write ratio is high.
    • Propagate the update operation to others
      • Less network bandwidth, but more processing overhead.
**Pull vs. Push**

- Push-based (server-based) approach: updates are propagated to other replicas by the server
  - Used in permanent and server-initiated replicas, can maintain high degree of consistency.
  - Good for high read-to-update ratio.
  - The server needs to keep track of the clients.
- Pull-based (client-based) approach: client checks with the server whether the cached data is still up-to-date.
  - Good for low read-to-update ratio, but high response time.

A comparison in multiple client, single server systems.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Push-based</th>
<th>Pull-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>State at server</td>
<td>List of client replicas and caches</td>
<td>None</td>
</tr>
<tr>
<td>Messages sent</td>
<td>Update (and possibly fetch update later)</td>
<td>Poll and update</td>
</tr>
<tr>
<td>Response time at client</td>
<td>Immediate (or fetch-update time)</td>
<td>Fetch-update time</td>
</tr>
</tbody>
</table>

**Leasing**

- A lease is a promise by the server that it will push updates to the client for a specified time. When a lease expires, the client is forced to poll the server for updates and pull the data if necessary.
- Age-based lease: give long-leasing time to data that remain unchanged for a long time.
- Give long leasing time to clients whose cache often needs to be accessed, and short lease for item which rarely accessed.
- Lower leasing time when the server is overloaded.

- Unicast vs. multicast. Multicast can be efficiently combined with push-based approach.

**Epidemic Protocols**

- Epidemic algorithms are based on the theory of epidemics, which studies the spreading of infectious diseases. Used to propagate the updates. A server can be:
  - Infective: if it holds an update that it is willing to spread to others.
  - Susceptible: has not yet been updated.
  - Removed: an updated server that is not willing to spread the update.
- P picks another server Q at random to exchange updates,
  - P only pushes its own updates to Q
  - P only pulls in new updates from Q
  - P and Q send updates to each other (i.e., push-pull)

**Epidemic Protocols**

- Only pushing updates is not enough to rapidly spreading updates.
  - Updates are propagated by infective servers. If many servers are infective, the probability of selecting a susceptible server is small.
- Pull-based approach works better when many servers are infective. In this case, spreading updates is triggered by susceptible servers.
- Rumor spreading (gossiping): if P has an update, it contacts Q. If Q has already been updated by others, P may lose interest and only spread the update with a probability of $1/k$.
- Some technique, such as directly pushing an update to a number of servers helps.
Removing Data

- Epidemic algorithms are good for spreading updates, not for deletion.
  - A delete removes all information about the data. Later, when it receives old copies of the data, it may add the data again.
- Solution is to treat a delete request as a death certificate, and keep this certificate.
  - How to clean up these certificates; otherwise, it wastes space.
- Solution: to timestamp the certificates. Since an update propagates to all servers within a finite time. The certificate can be removed after this time. To provide hard guarantee, some servers maintain some certificates. Spread the certificate again when seeing another update.

Consistency Protocols

- A consistency protocol describes an implementation of a specific consistency model
- Primary-based protocols
  - Each data item has an associated primary, which coordinates the write operation. The primary can be at a remote server or moving the primary to local
- Replicated write protocols
  - Writes can be carried out at multiple replicas
  - Active replication or majority voting

The principles of a primary-backup protocol (blocking).
- Non-blocking has good performance, but has problems with fault-tolerance

Primary-backup protocol in which the primary migrates to the process wanting to perform an update.
Replicated-Write: Active Replication

- Since multiple replicas can do update, operations need to ensure total order. Use Lamport’s clock or a sequencer.
- Forward each operation to the sequencer, which assigns a unique sequence number and then forwards to the replicas. Operations are carried out in the order of their sequence number. Problem: replicated invocations.

Solution: use coordinator.

Total Order

- Each message is timestamped, and multicasted to other processes (and itself).
- When a process receives a message, put it into a local queue, and multicast an ack to other processes.
- A process can deliver a queued message only when the message is at the head of the queue and has been acked by each other process.
  - At this time, deliver the message and remove the acks.

Continuous Consistency: Numerical Errors

- Every server $S_i$ has a log, denoted as $L_i$.
- Consider a data item $x$ and let $\text{val}(W)$ denote the numerical change in its value after a write operation $W$. Assume that
  \[
  \forall W : \text{val}(W) > 0
  \]
- $W$ is initially forwarded to one of the $N$ replicas, denoted as $\text{origin}(W)$. $TW[i,j]$ are the writes executed by server $S_j$ that originated from $S_i$:
  \[
  TW[i,j] = \sum \{ \text{val}(W) | \text{origin}(W) = S_j \text{ & } W \in L_i \} 
  \]
Continuous Consistency: Numerical Errors

\[ v = v_0 + \sum_{k=1}^{N} TW[k,k] \]
\[ v_i = v_0 + \sum_{k=1}^{N} TW[i,k] \]
Note that \( v_i \leq v \)

- We need to ensure that for every server \( S_i \)
  \[ v - v_i \leq \delta_i \]

Let every server \( S_k \) maintain a view \( TW_k[i,j] \) of what it believes is the value of \( TW[i,j] \). This information can be gossiped when an update is propagated.

\[ 0 \leq TW_k[i,j] \leq TW[i,j] \leq TW[j,j] \]

\( S_k \) sends operations from its log to \( S_i \) when it sees that \( TW_k[i,k] \) is getting too far from \( TW[k,k] \), in particular, when

\[ TW[k,k] - TW_k[i,k] > \delta_i/(N - 1) \]

Case Studies

File

- A file is a sequence of bytes
- A file can have attributes, which are information about the file such as the owner, size, creation data, and access permissions
  - In UNIX, these attributes and the file addresses are saved in the inode.
- A file service usually provides primitives to read and write some of the attributes and the data
  - The upload/download model: files moved to clients. Clients do all the operations and send it back.
  - The remote access model: files stay on the server, and the server do all the modifications.

Distributed File Systems

- In a distributed file system, files can be stored at any machine.
- For higher performance, several machines, referred to file servers, are dedicated to storing files and performing storage and retrieve operations.
- Name server: a process that maps names specified by clients to stored objects such as files and directories.
- Cache manager: a process that implements file caching
  - Can be present at both clients and servers.
**Replication**

- Replicate the file system can improve availability and performance.
- Replication may generate some problems
  - How to keep replicas of a file consistent?
  - How to detect and recover inconsistencies among replicas?
- Solutions: primary copy replication
  - One server is designated as primary, and others are secondaries. The change is made to the primary, which is responsible for propagating to secondaries.
  - What happens when the primary fails?
  - Other solutions such as voting.

**Replication Approaches**

- Explicit file replication, use the whole path to copy files to different sites
- Lazy replication, replication happens in the background
- Using a group, all copies are made at the same time

**Semantics of File Sharing**

- UNIX semantics
  - Every operation on a file is instantly visible to all processes. Poor performance.
- Session semantics
  - No changes are visible to other processes until the file is closed.
- Immutable files
  - No updates are possible, simplifies sharing and replication.
  - What to do if a file is replaced while another process is reading it. Solution: read the old one or let it fail.
- Transactions
  - All changes have the all-or-nothing property.

(a) On a single processor, when a read follows a write, the value returned by the read is the value just written.

(b) In a distributed system with caching, obsolete values may be returned.
Where to Cache

- The server’s disk
  - Plenty of space
  - Performance is poor
- The server’s memory
  - Cache unit: whole file or disk blocks
  - Cache replacement algorithm
  - Consistency is easy to maintain and transparent to the client
- Client’s memory or client’s disk

Page 57

Where to Cache

- Advantages of cache on main memory
  - Diskless workstations can also use caching
  - Accessing a cache in main memory is faster
- Disadvantages:
  - It competes with the virtual memory system for physical memory space. Need more complex system
  - Large files cannot be cached completely
- Advantages of cache on disk
  - Large files can be cached
  - Virtual memory management is simple
  - Can be used by portable computers
- Broadcast disks

Page 58

Write Through

- Client side caching introduces inconsistency
  - Two clients simultaneously read the same file and modify the file. What will the third client see?
- Write through algorithm: when a cache entry (file or block) is modified, the new value is kept in the cache, but is also immediately written to the server.
- Problems:
  - Client on A reads f, and keeps f in its cache.
  - Later, client on B reads the same file, modifies, and writes through to the server.
  - A new client on A starts and reads f from its cache.
- Solution: Let the client check the cache validity with the server.

Page 59

Delayed Write

- Write through helps read, not write.
- Solution: delayed writing policy.
- Delays the writing at the server
  - All the file updates are gathered together and sent to the server all at once.
  - Many files are deleted after a short time, e.g., compilers generate some files.
- Problems: consistency, reliability, tradeoff between performance and cleaner semantics.

Page 60
Other Approaches

- Write-on-close
  - Write a file back to the server only after it has been closed.
  - Wait some time after the close to see if it is deleted.
  - Matches session semantics.
- Centralized control
  - The file server keeps track of who is accessing which file.
  - If a file is open for reading, no one can write.
  - If a file is open for write, no one can write and read.
  - Not scalable and suffer from single point failure.

NFS

- Network file system (NFS) was developed by SUN Microsystems.
- It provides transparent access to remote files for client programs running on UNIX and other systems.
- Each computer has a client and server modules installed in its system kernel.
- Achieve a high level of support for hardware and OS heterogeneity.

Virtual File System

- The access transparency is achieved by a virtual file system (VFS) module.
- VFS keeps track of the file systems that are currently available both locally and remotely, and it passes each request to the appropriate system.
- V-node contains a reference to show whether a file is local or remote
  - If the file is local, the v-node contains a reference to the index of the local file (i-node).
  - If the file is remote, it contains the file handle of the remote file. The file handle contains fields uniquely identifying the file system type, the disk, the i-node number of the directory, and security information.
Mount Services

- On each server, a file /etc/exports contains the names of local file systems that are available for remote mounting.
  - An associated access list indicates which hosts are permitted to mount.
  - By using the mount protocol, a client can send a path name to the server and request permissions to mount that directory somewhere in its directory hierarchy.
- Remote file systems may be hard-mounted or soft-mounted
  - If the accessed file is hard-mounted, the process is suspended until the request is completed.
  - In case of soft-mounted, the NSF returns a failure indication to user-level processes after some retries.

Remount Mount

![Diagram showing Mount Services]

Automounting

- Clients contain a file called /etc/rc, which contains the remote mount commands.
- NFS supports automounting as follows
  - A client has a set of remote directories to be associated with a local directory.
  - None of them are mounted when the client is booted.
  - The first time a remote file is opened, the OS sends a message to each of the servers.
  - The first one to reply wins, and its directory is mounted.
- Advantages of automounting
  - Fault-tolerant
  - High performance
- Assumptions: all file systems are identical, usually for read-only file systems.

Client Cache

- Clients are responsible for validating its cache with the server.
- A timestamp-based approach is used to validate local cache
  - $T_c$ is the time when the cache entry was last validated.
  - $T_m$ is the time when the block was last modified at the server.
  - Condition for valid:
    $$ (T - T_c < t) \lor (T_m^{\text{client}} = T_m^{\text{server}}) $$
    - $t$ is 3-30 seconds for individual files based on the update frequency.
    - $t$ is 30-60 seconds for directories.
**Coda**

- Coda (constant data availability) file system is a descendant of the Andrew file system (AFS) developed at CMU.
- Design goals of Coda: scalability and constant data availability
  - Replicate servers
  - Only depends on local cache when no server can be contacted
- The unit of replication is volume, which consists of a set of files and directories located on a server
  - Each file and directory is identified by a 92-bit-long file identifier (FID).

**Cache and Replication**

- When a volume is created, the number of replicas and the servers that will store these replicas are specified.
- The set of servers storing the replicas of a volume constitutes its volume storage group (VSG).
- The set of servers that are accessible to a client is called the accessible volume storage group (AVSG).
- The AVSG for the cached volume is kept track of by Venus (the client cache manager)
  - At a cache miss, a client obtains data from a preferred server (one of the AVSG).
  - A preferred server can be chosen randomly or based on server load, distance.
  - The client also verifies with other servers if the preferred server does indeed have the latest copy. If not, get latest copy from other servers and notify the existence of stale data.

**Callback**

- The callback is a server-initiated approach to maintain cache consistency.
- A callback is a promise by the server that it will notify the client if the file is modified by some other client.
- Receiving a notification, the client invalidates the cached data and reestablishes a callback upon fetching the data.
- Due to network failure, a notification may be lost.
- Coda uses an optimistic strategy for updates, and then can be used in mobile environments.

**Cache Coherence**

- The Venus continuously monitors the cache validity.
- **Enlargement of the AVSG**
  - Since some inaccessible server can become accessible, some cached data may become invalid.
  - In such case, Venus cancels previous callbacks related to the invalid data.
  - The next reference to the data requires refetch and reestablish callbacks.
- **Shrinking of the AVSG**
  - Some servers become inaccessible.
  - If the shrinking is caused by the preferred server, the callbacks are canceled.
- Venus detects the enlargement and shrinking of the AVSG by probing the members of VSG every 10 minutes.
**Lost Callbacks**

- Callbacks are established only at the preferred server.
- Problems
  - Since the preferred server of one client need not be in the AVSG of another client, updates by the second client may not result in callbacks on the first client.
- Solution
  - Each probe by Venus requests a *volume coda version vector* (VCVV) for every volume from which it has cached data.
  - The VCVV is updated at every modification of the volume. Lost updates are indicated by a mismatch in VCVVs at the client and at the server, in which case, Venus cancels callbacks on the associated cached data.

**Replica Management**

- Coda allows modification of files to proceed when the network is partitioned.
- It relies on coda version vector (CVV) to identify conflicts
  - A CVV is a vector timestamp with one element for each server in the relevant VSG.
  - Each element is an estimate of the update version number of the corresponding server.
  - If neither \( v_1 \geq v_2 \) nor \( v_2 \leq v_1 \) holds, a conflict.
- After a modified file is closed, Venus sends an update to each site in AVSG, including the current CVV
  - Receiving the update message, each server returns a positive ACK.
  - Venus calculates new CVV with modification counts increased for the servers that responded positively, and distributes it to members in AVSG.

Assume F is replicated at 3 servers S1, S2, S3. The VSG for F is \{S1, S2, S3\}. F is modified at the same time by clients C1 and C2. C1’s AVSG is \{S1, S2\}, C2’s AVSG is \{S3\}.

1. Initially, the CVV for F at all 3 servers are \([1,1,1]\)
2. C1 opens, modifies, and closes F. The Venus at C1 broadcasts an update message to its AVSG \{S1, S2\}, finally resulting in a new version of F and a CVV \([2,2,1]\) at S1 and S2, but no change at S3.
3. Meanwhile, C2 runs two processes, each of which modifies and closes F, resulting in a CVV \([1,1,3]\) at S3.
4. When the network is reconnected, C2 makes routine check and discovers that S1 and S2 are now accessible. It modifies its AVSG to \{S1,S2, S3\} for F and requests the CVVs for F from all members of AVSG. Then, it finds the conflict.

On the other hand, if (3) does not exist, the CVV at S3 is still \([1,1,1]\), and C2 discovers that the CVV at S1 and S2 \([2,2,1]\) dominates that at S3. The version of the file at S1 or S2 should replace that at S3.

**Disconnected Operations**

- Users specify a prioritized list of files and directories that Venus should try to retain in the cache.
- During disconnection, AVSG is empty.
- When disconnected operations end, a process of reintegration begins. Venus executes a sequence of update operations to make the AVSG replicas identical to the cached copy.
- Conflicts may be detected during reintegration due to updates to AVSG replicas by other clients.
  - Coda assigns priority to server-based replicas over cached copies.
Replication in Peer-to-Peer networks

- In P2P, all files are read only. Updates consist only in the form of adding files to the system.
- In unstructured P2P systems, how to distribute the $n$ copies.
  - Uniformly distribute
  - Replicate files according to how often they are searched for: the more popular it is, the more replicas we create.
  - Side effect: very expensive to locate unpopular ones.

Structured P2P Systems

- Replicate a file along the path that a query has followed from source to destination. Then, most replicas will be close to the original source and reduce its load.
- Does not take the load of other nodes into account.
- Solution: $R$ checks if any of its files should be offloaded to $P$.

Consistency and Replication in the Web

- User sets browser: Web accesses via cache
- Browser sends all HTTP requests to cache
  - Object in cache: cache returns object
  - Else cache requests object from origin server, then returns object to client

Web Proxy Caching

- Various caches: cache provided by the browser. Web proxy provided by an organization.
- Hierarchical cache: different levels of organizations all provide proxy caches, form a hierarchy. Although it can reduce the network traffic, the delay also increases.
- Cooperative cache: neighboring proxies cooperate with each other. Effective for a small groups of clients.
Cache Consistency

- Use conditional GET, but the proxy needs to contact the server for each request.
- Weaker consistency is used in the Squid Web proxy.
  \[ T_{\text{expire}} = \alpha (T_{\text{cached}} - T_{\text{last modified}}) + T_{\text{cached}} \]
  With \( \alpha = 0.2 \), Until \( T_{\text{expire}} \), the data is valid.
- Although it can reduce the network traffic, clients may get stale data.
- Invalidation-based approach: the server sends invalidation to clients.
- Cache can only be used for static data.

Conditional GET

- **Goal:** don’t send object if cache has up-to-date cached version
- **cache:** specify date of cached copy in HTTP request
  
  - If-modified-since: \(<\text{date}>\)

- **server:** response contains no object if cached copy is up-to-date:
  
  - HTTP/1.0 304 Not Modified

- Invalidation-based approach: the server sends invalidation to clients.
- Although it can reduce the network traffic, clients may get stale data.
- Cache can only be used for static data.

Replication for Web Hosting Systems

- Uncontrollable parameters (disturbance / noise)
- Initial configuration
- Connections
- Web hosting system
- Observed output
- Reference input
- Metric estimation
- Measured output

- The general organization of a CDN as a feedback-control system

Metric Estimation

- Latency to get a document.
- Spatial metric which measures the distance between nodes in terms of the number of hops.
- Network usage, the consumed bandwidth
- Consistency metric, how stale the data is.
- Financial metric,
- These metrics are hard to measure. In practice, the most important issue is whether CDN can satisfy the service-level agreement.
**Adaptation Triggering**

- When and how adaptation should be triggered. Periodic?
- Deal with flash crowd, which is a sudden burst in requests for a specific web document.
  - Replicate the website
  - By predicting the coming of flash crowd, install replicas and redirect the requests.
- Linear extrapolation technique: continuously measure the number of requests to a document during a specific time interval (window/slots). By applying linear regression, we can fit a curve expressing the number of accesses as a function of time, and can be used to predict.

---

- One normal and three different access patterns reflecting flash crowd behavior

---

**Adjustment Measures (client-request redirection)**

---

**Caching Dynamic Content**

- Up to this point, we have mainly concentrated on caching static content. For dynamic content, the edge server can replicate the application.
  - For full replication, the read to update ratio should be high, or the query is very complex such as join.
  - If most queries only access a single table, partial replication is enough.
- Content-aware cache: edge server maintains a tailored database. Queries adhere to a limited number of templates can be processed at the edge server; others will be sent to the original server. The server needs to know which records are associated with which template.
- Content-blind caching: receiving a query, the edge server computes a unique hash value, and check if it has processed this query before.
Replication of Web Applications

Cache Consistency

- Server-initiated approach
  - The server informs cache managers whenever the data in the client caches become stale.
  - Client cache managers get new data from the server.
- Client-initiated approach
  - Clients verify with the server before each use.
- Both are not scalable
- Invalidation Report (IR) based approach
  - The server periodically broadcasts a report in which the data items that have been updated are broadcasted.
  - Clients only need to wake up at the IR broadcast time, and use the IR to invalidate their caches.

The IR-Based Caching Strategy

- The server constructs and broadcasts IR
  - \( IR_i = \{ <d_x, t_x> \mid (d_x \in D) \land (T_i - T_{i-w} < t_x \leq T_i) \} \)
- When a client receives an IR
  - if disconnected too long
  - then drop the entire cache or go uplink to verify the cache
  - for each data item \( d_x \) in IR
  - invalidate \( d_x \) if it is outdated.
- Other approaches construct different IRs to deal with the long disconnection problem.

Enhancement

- Reduce the timestamp overhead
  - Use \( <T_k, D> \) to represent those items that have been updated in the \((k-1)th\) interval.
- In this way, the client does not need to listen to all IR reports
  - Suppose the window size is \( w \). If the client did not miss the report \( IR_{i-1} \), it only needs to download the last \( <T_i,D> \) instead of from \( <T_{i-1},D> \) to \( <T_{i-w},D> \).