DC TT2 ANSWERKEY

Chap 4

1. Desirable features of a Global Scheduling Algorithm.

i. General purpose:

- ✓ A scheduling approach should be uniformly applicable to different types of applications that can be executed.
- ✓ Interactive jobs, distributed and parallel applications, as well as non-interactive batch jobs, should all be supported with good performance.
- ✓ Achieving this is difficult because different kinds of jobs have different attributes, their requirements to the scheduler may contradict. To achieve the general purpose, a trade-off may have to be made.

ii. Efficiency:

- ✓ Should improve the performance of scheduled jobs as much as possible.
- ✓ Scheduling should incur reasonably low overhead so that it would not counter attack the benefits.

iii. Fairness:

- ✓ Sharing resources among users raises new challenges in guaranteeing that each user obtains his/her fair share when demand is heavy.
- ✓ In a distributed system, this problem could be worsened such that one user may consume the entire system.
- ✓ This may lead to Starvation or DNS

iv. Dynamic:

- ✓ The algorithms employed to decide where to process a task should respond to load changes and exploit the full extent of the resources available.

v. Transparency:

- ✓ The behavior and result of a task's execution should not be affected by the host(s) on which it executes.
- ✓ There should be no difference between local and remote execution.
- ✓ No user effort should be required in deciding where to execute a task or in initiating remote execution.
- ✓ Further, the applications should not be changed greatly.
- ✓ It is undesirable to have to modify the application programs in order to execute them in the system.

vi. Scalability:

- ✓ A scheduling algorithm should scale well as the number of nodes increases.
- ✓ An algorithm that makes scheduling decisions by first inquiring the workload from all the nodes and then selecting the most lightly loaded node has poor scalability.
- ✓ This will work fine only when there are few nodes in the system. Also, the network traffic quickly consumes network bandwidth. A simple approach is to probe only m of N nodes for selecting a node.

vii. Fault tolerance:

- ✓ A good scheduling algorithm should not be disabled by the crash of one or more nodes of the system.
- ✓ Also, if the nodes are partitioned into two or more groups due to link failures, the algorithm should be capable of functioning properly for the nodes within a group.
- ✓ Algorithms that have decentralized decision making capability and consider only available nodes in their decision making have better fault tolerance capability.

viii. Quick decision-making capability:

- ✓ Heuristic methods requiring less computational efforts (and hence less time) while providing near-optimal results are preferable to exhaustive (optimal) solution methods.

ix. Balanced system performance and scheduling overhead:

- ✓ Algorithms that provide near-optimal system performance with a minimum of global state information gathering overhead are desirable. This is because, the overhead increases as the amount of global state information collected increases.

x. Stability:

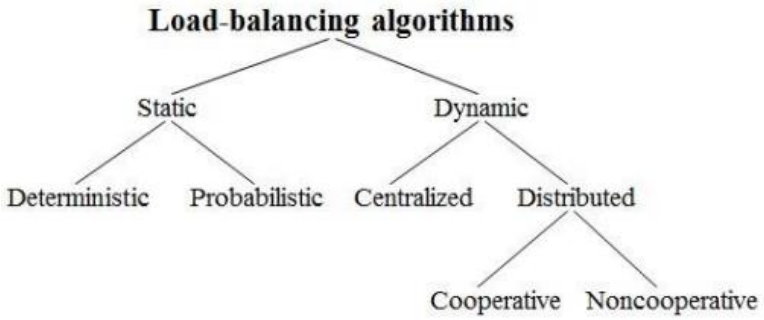
- ✓ Fruitless migration of processes, known as processor thrashing, must be prevented.
- ✓ E.g. if nodes n1 and n2 observe that node n3 is idle and then offload a portion of their work to n3 without being aware of the offloading decision made by the other node. Now if n3 becomes overloaded due to this it may again start transferring its processes to other nodes.
- ✓ This is caused by scheduling decisions being made at each node, independent of decisions made by other nodes.

2. Task assignment approach

- ✓ The following assumptions are made in this approach:
  - A process has already been split up into pieces called tasks.
  - This split occurs along natural boundaries (such as a method), so that each task will have integrity in itself and data transfers among the tasks are minimized.
  - The amount of computation required by each task and the speed of each CPU are known. The cost of processing each task on every node is known.
  - The IPC cost between every pair of tasks is already known.
  - The IPC cost is 0 for tasks assigned to the same node.
  - If two tasks communicate n times and the average time for each inter-task communication is t, then IPC costs for the two tasks is n * t.
  - Precedence relationships among the tasks are known.
  - Reassignment of tasks is not possible.
- ✓ **Goal** of this method is to assign the tasks of a process to the nodes of a distributed system in such a manner as to achieve goals such as:
  - Minimization of IPC costs.
  - Quick turnaround time for the complete process.
  - A high degree of parallelism.
  - Efficient utilization of system resources in general.
- ✓ These goals often conflict.
  - E.g., while minimizing IPC costs tends to assign all tasks of a process to a single node, efficient utilization of system resources tries to distribute the tasks evenly among the nodes.
- ✓ So also, quick turnaround time and a high degree of parallelism encourage parallel execution of the tasks, the precedence relationship among the tasks limits their parallel execution.
- ✓ In case of m tasks and q nodes, there are m*q possible assignments of tasks to nodes.
- ✓ In practice, however, the actual number of possible assignments of tasks to nodes may be less than m*q due to the restriction that certain tasks cannot be assigned to certain nodes due to their specific requirements

3. Load balancing approach

- ✓ A load balancer is a device that acts as a reverse proxy and distributes network or application traffic across a number of servers.
- ✓ The load balancer is a framework that can deal with the load and is utilized to disperse the assignments to the servers.
- ✓ The load balancers allocate the primary undertaking to the main server and the second assignment to the second server.



- ✓ **Purpose of Load Balancing in Distributed Systems:**
  - **Security:** A load balancer provide safety to your site with practically no progressions to your application.
  - **Protect applications from emerging threats:** The Web Application Firewall (WAF) in the load balancer shields your site.
  - **Authenticate User Access:** The load balancer can demand a username and secret key prior to conceding admittance to your site to safeguard against unapproved access.
  - **Protect against DDoS attacks:** The load balancer can distinguish and drop conveyed refusal of administration (DDoS) traffic before it gets to your site.
  - **Performance:** Load balancers can decrease the load on your web servers and advance traffic for a superior client experience.
  - **SSL Offload:** Protecting traffic with SSL (Secure Sockets Layer) on the load balancer eliminates the upward from web servers bringing about additional assets being accessible for your web application.
  - **Traffic Compression:** A load balancer can pack site traffic giving your clients a vastly improved encounter with your site.

✓ Static versus Dynamic

Static	Dynamic
Static algorithms use only information about the average behavior of the system.	Dynamic algorithms collect state information and react to system state if it changed.
Static algorithms ignore the current state or load of the nodes in the system.	Dynamic algorithms are able to give significantly better performance.
Static algorithms are much simpler.	They are complex

✓ Deterministic versus Probabilistic

Deterministic	Probabilistic
Deterministic algorithms use the information about the properties of the nodes and the characteristic of processes to be scheduled.	Probabilistic algorithms use information of static attributes of the system (e.g. number of nodes, processing capability, topology) to formulate simple process placement rules
Deterministic approach is difficult to optimize.	Probabilistic approach has poor performance

✓ Centralized versus Distributed

Centralized	Distributed
Centralized approach collects information to server node and makes assignment decision.	Distributed approach contains entities to make decisions on a predefined set of nodes
Centralized algorithms can make efficient decisions, have lower fault-tolerance	Distributed algorithms avoid the bottleneck of collecting state information and react faster

✓ Cooperative versus Non-cooperative

Cooperative	Non-cooperative
In Co-operative algorithms distributed entities cooperate with each other.	In Non-cooperative algorithms entities act as autonomous ones and make scheduling decisions independently from other entities.
Cooperative algorithms are more complex and involve larger overhead.	They are simpler
Stability of Cooperative algorithms are better.	Stability is comparatively poor.

✓ Issues in Load Balancing Algorithms
- Load estimation policy: determines how to estimate the workload of a node.
- Process transfer policy: determines whether to execute a process locally or remote.
- State information exchange policy: determines how to exchange load information among nodes.
- Location policy: determines to which node the transferable process should be sent.
- Priority assignment policy: determines the priority of execution of local and remote processes.
- Migration limiting policy: determines the total number of times a process can migrate

4. Load Sharing Approach

✓ The following problems in load balancing approach led to load sharing approach:
- Load balancing technique with attempting equalizing the workload on all the nodes is not an appropriate object since big overhead is generated by gathering exact state information.
- Load balancing is not achievable since number of processes in a node is always fluctuating and temporal unbalance among the nodes exists every moment.

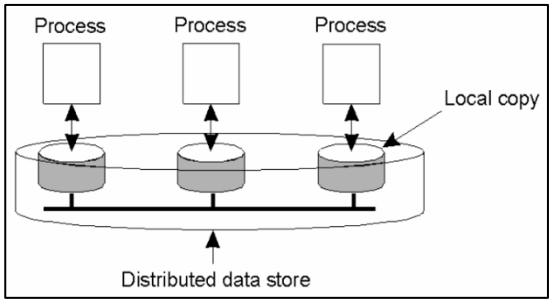
✓ **Basic idea:**
- It is necessary and sufficient to prevent nodes from being idle while some other nodes have more than two processes.
- Load-sharing is much simpler than load-balancing since it only attempts to ensure that no node is idle when heavily node exists.
- Priority assignment policy and migration limiting policy are the same as that for the load-balancing algorithms.

- ✓ **Load Estimation Policies**
  - Since load-sharing algorithms simply attempt to avoid idle nodes, it is sufficient to know whether a node is busy or idle.
  - Thus, these algorithms normally employ the simplest load estimation policy of counting the total number of processes.
  - In modern systems where permanent existence of several processes on an idle node is possible, algorithms measure CPU utilization to estimate the load of a node.
- ✓ **Process Transfer Policies**
  - The load sharing algorithms normally use all-or-nothing strategy.
  - This strategy uses the threshold value of all the nodes fixed to 1.
  - Nodes become receiver node when it has no process, and become sender node when it has more than 1 process.
  - To avoid processing power on nodes having zero process load-sharing algorithms uses a threshold value of 2 instead of 1.
  - When CPU utilization is used as the load estimation policy, the double-threshold policy should be used as the process transfer policy
- ✓ **Location Policies**
  - The location policy decides whether the sender node or the receiver node of the process takes the initiative to search for suitable node in the system, and this policy can one of the following:
    - Sender-initiated location policy: Sender node decides where to send the process. Heavily loaded nodes search for lightly loaded nodes
    - Receiver-initiated location policy: Receiver node decides from where to get the process. Lightly loaded nodes search for heavily loaded nodes
- ✓ **Sender-initiated location policy**
  - Node becomes overloaded, it either broadcasts or randomly probes the other nodes one by one to find a node that is able to receive remote processes.
  - When broadcasting, suitable node is known as soon as reply arrives.
- ✓ **Receiver-initiated location policy**
  - Nodes becomes underloaded, it either broadcast or randomly probes the other nodes one by one to indicate its willingness to receive remote processes.
  - Receiver-initiated policy require preemptive process migration facility since scheduling decisions are usually made at process departure epochs
    - Both policies give substantial performance advantages over the situation in which no load-sharing is attempted.
    - Sender-initiated policy is preferable at light to moderate system loads.
    - Receiver-initiated policy is preferable at high system loads.
    - Sender-initiated policy provide better performance for the case when process transfer cost significantly more at receiver-initiated than at sender-initiated policy due to the pre-emptive transfer of processes.
- ✓ **State information exchange policies**
  - In load-sharing algorithms it is not necessary for the nodes to periodically exchange state information, but needs to know the state of other nodes when it is either underloaded or overloaded.
  - The following are the two approaches followed when there is state change:
    - Broadcast when state changes
    - In sender-initiated/receiver-initiated location policy a node broadcasts State Information Request when it becomes overloaded/ underloaded.
    - It is called broadcast-when-idle policy when receiver-initiated policy is used with fixed threshold value of 1
      - Poll when state changes
    - In large networks polling mechanism is used.
    - Polling mechanism randomly asks different nodes for state information until find an appropriate one or probe limit is reached.
    - It is called poll-when-idle policy when receiver-initiated policy is used with fixed threshold value of 1.

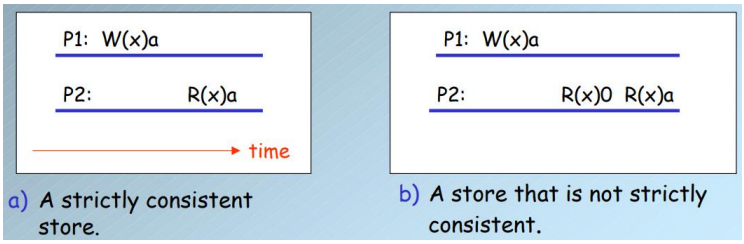
Chap 5

- ✓ **Consistent Ordering of Operations**
  - Besides continuous consistency, there is a huge body of work on datacentric consistency models from the past decades.
  - An important class of models comes from the field of concurrent programming.
  - Researchers have sought to express the semantics of concurrent accesses when shared resources are replicated.
  - This has led to at least one important consistency model that is widely used. In the following, we concentrate on what is known as Strict consistency, sequential consistency, Linearizability, causal consistency, FIFO, Weak, Release, and Entry.
- ✓ A consistency model is a contract between a DS data-store and its processes.
- ✓ Since it is difficult to define precisely which write operation is the last one.
- ✓ If the processes agree to the rules, the data-store will perform properly and as advertised

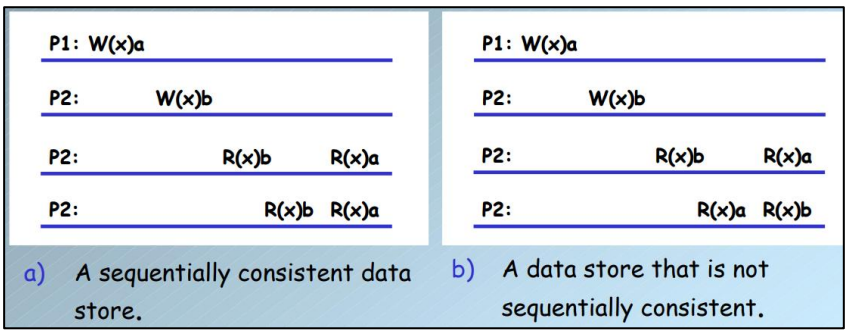
1. Data Centric Consistency Models (Strict consistency, sequential consistency, Linearizability, causal consistency, FIFO)



- ✓ Data store: – A term used to refer to a physically distributed shared data. A data-store can be read from or written to by any process in a DS.
- ✓ A local copy of the data-store (replica) can support "fast reads".
- ✓ However, a write to a local replica needs to be propagated to all remote replicas.
- ✓ **Strict Consistency**
  - Any read on a shared data item x returns a value corresponding to the result of the most recent write on x.
  - All writes are instantaneously visible to all processes
  - It is impossible to implement it in distributed systems since strict consistency relies on absolute global time.



- ✓ **Sequential Consistency**
  - When processes run concurrently on (possibly) different machines, any valid order of read and write operations is acceptable, as long as all processes observe the same order of operations
  - No reference to "most recent" write
  - No reference to physical time
  - No reference to logical time
  - Somewhat a weaker consistency model



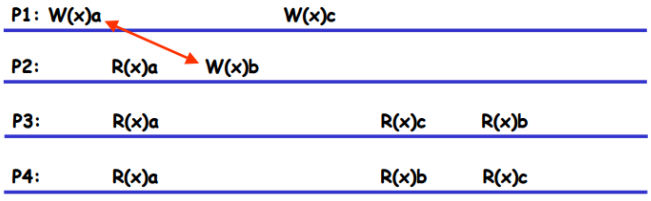
- ✓ **Linearizability**
  - It is similar to sequential consistency; but stronger than sequential consistency (but weaker than strict consistency).
  - Operations are assumed to receive a timestamp using a globally available clock – one with finite precision.
  - If $ts_{op1}(x) < ts_{op2}(x)$ then OP1(x) should precede OP2(x) in the sequence.

- Linearizable data is also sequentially consistent.
- Processes use loosely synchronized clocks.
- Example:
  - Three concurrently executing processes: P1, P2, P3.
  - Initial setting: x=y=z=0
  - Assignment → write
  - print →read
  - With six operations, there are 6! = 720 possible execution sequences, (however, some of these violate program order)
  - Only, 90 of them preserve the program order.

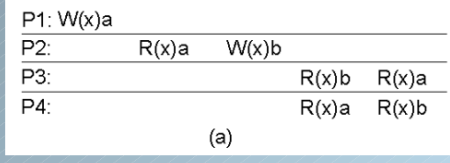
Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

✓ **Causal consistently**

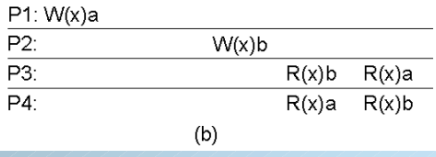
- Necessary condition:
  - ➢ Writes that are potentially casually related must be seen by all processes in the same order.
  - ➢ Concurrent writes (that are not causally related) may be seen in a different order on different machines.
- Two writes are causally related to each other through a read operation.
- Vector timestamps are used to implement casual consistency.
- Example:



- This sequence is allowed with a casually-consistent store, but not with sequentially or strictly consistent store.



a) A violation of a casually-consistent store.



b) A correct sequence of events in a casually-consistent store.

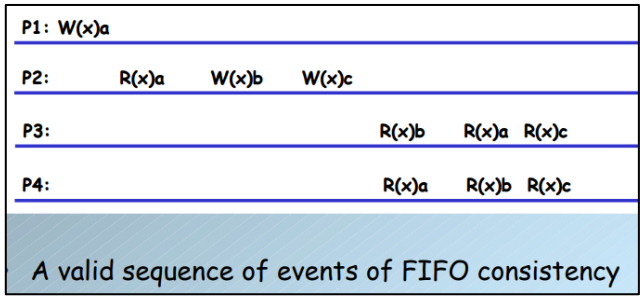
✓ **FIFO Consistency:**

- Necessary Condition:
  - Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.
- Weaker form of strong consistency.
- All writes generated by different processes are concurrent.

P1:	P2:	P3:
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

x = 1;	x = 1;	y = 1;
print (y, z);	y = 1;	print (x, z);
y = 1;	print(x, z);	z = 1;
print(x, z);	print (y, z);	print (x, y);
z = 1;	z = 1;	x = 1;
print (x, y);	print (x, y);	print (y, z);
Prints: 00	Prints: 10	Prints: 01
(a)	(b)	(c)

Statement execution as seen by the three processes from the previous slide. The statements in bold are the ones that generate the output shown.



A valid sequence of events of FIFO consistency

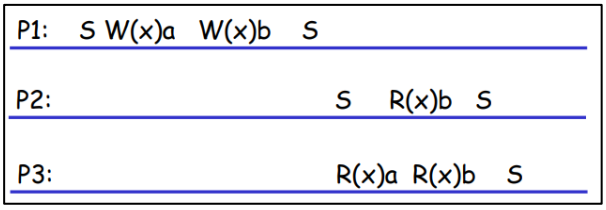
Process P1	Process P2
x = 1;	y = 1;
if (y == 0) kill (P2);	if (x == 0) kill (P1);

- Two concurrent processes.
- Both processes can be killed in FIFO consistency

2. Grouping Operations (Weak, Release, Entry)

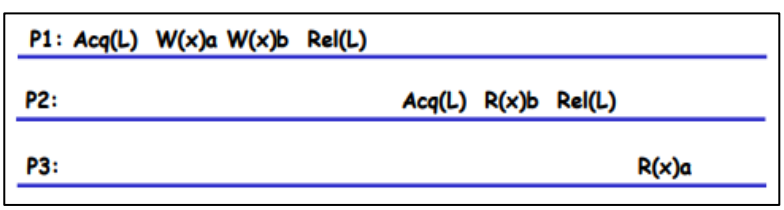
✓ **Weak Consistency Model**

- Basic idea:
  - Individual read & write operations are not immediately made known to other processes.
  - Final effect is communicated (as in transactions)
  - synchronization variable (S) (a lock or barrier)
  - A synchronization variable has only one operation: synchronize(S)
  - A process gains exclusive access to a critical region through synchronization operation
  - While a process is in the critical region, the inconsistencies will happen.
  - *synchronize* operation pushes local updates to other replicas + brings about the remote updates to the local replica.
- Properties
  - Access to synchronization variable is sequential
  - ➢ If processes P1 and P2 call synchronize(S), the execution order of these operations will be the same everywhere.
  - Synchronization flushes the pipeline.
  - ➢ It forces all writes that are in progress or partially completed or completed at some local copies but not all to complete everywhere.
  - When data items are accessed, either for reading or writing, all previous synchronization will have been completed.
  - ➢ By doing synchronization before reading a shared data, a process can be sure of getting the most recent values.
  - A good consistency model when isolated accesses to shared data is rare.
  - With weak consistency, sequential consistency is enforced between groups of operations
  - Synchronization variables to delimit those groups.
  - Weak consistency models tolerate a greater degree of inconsistency for a limited amount of time.
- Example:



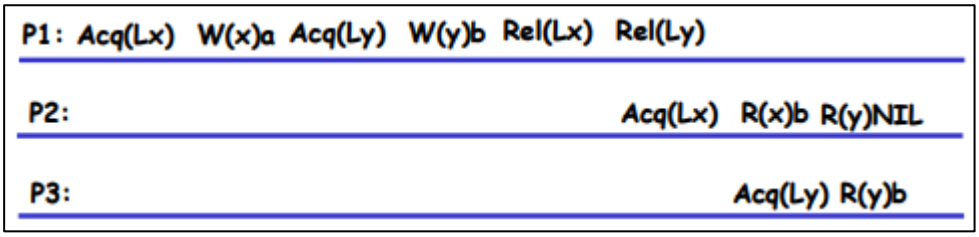
✓ **Release Consistency model**

- Drawbacks of weak consistency: When a synchronization variable is accessed by a process, the data store does not know if
  - the process is finished writing data (exiting critical region)
  - or it is just about to read data (entering critical region)
- Therefore, when an access to a synchronization variable is initiated, the local data store does two things:
  - All locally initiated writes have been propagated to all other copies
  - Gather in all writes from other copies
- If data store makes the distinction between entering or exiting from critical regions, problems will be solved
- Two types of synchronization operations are used:
  - Acquire
  - Release
- Programmer is responsible to call these operations before entering and exiting critical region



- Eager release consistency: release operation pushes out all the modified data to all other process
  - It does not matter whether the other processes need that data
- Lazy release consistency: release operation does send nothing
  - Acquiring process must come and get them.

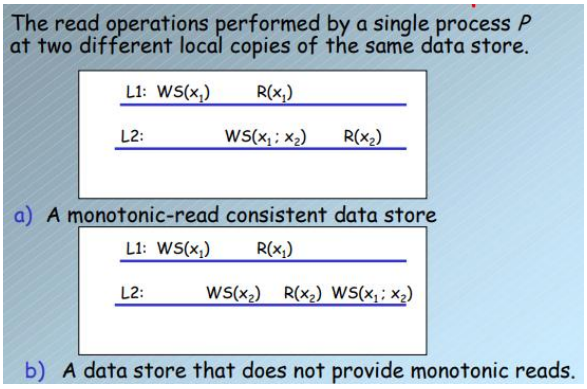
- ✓ **Entry Consistency Model**
  - With release consistency, all local updates are made available to all copies during the release of the lock
  - With entry consistency, each individual shared data item is associated with some synchronization variable (e.g. lock or barrier)
  - When acquiring the synchronization variable, the most recent values of its associated shared data item must be fetched
  - Note: where release consistency affects all shared data, entry consistency affects only those associated with a synchronization variable.
  - Conditions
    - At an acquire, all remote changes to the guarded data must be made visible
    - Before updating a shared data item, a process must enter the critical region in exclusive mode
    - If a process wants to enter a critical region in nonexclusive mode, it must first check with the owner of the synchronization variable to fetch the most recent copies of shared data.
    - Example:



3. Client-Centric Consistency Models (Eventual Consistency, Monotonic Reads, Monotonic Writes, Read Your Writes, Writes Follow Reads)

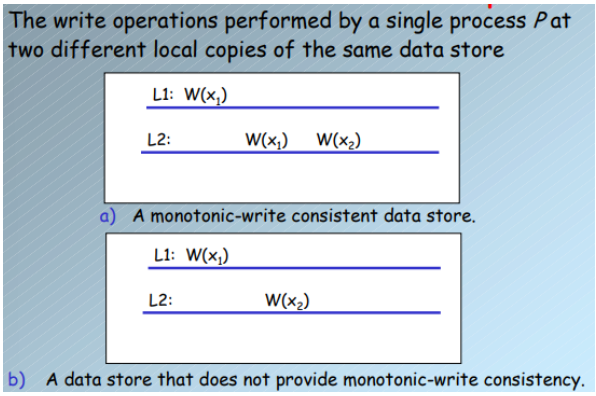
- ✓ Client-centric models provide guarantee of consistency for accesses of a single client
- ✓ Most large-scale distributed systems apply replication for scalability
- ✓ Simultaneous updates are rare; and if they happen, they are easy to resolve
- ✓ DNS: Updates are done by one process in a domain (no write-write conflicts); propagate slowly
- ✓ WWW: Caches all over the place, but there need be no guarantee that you are reading the most recent version of a page
- ✓ **Eventual Consistency:**
  - Large-scale distributed and replicated databases can tolerate a relatively high degree of inconsistency.
  - If no updates take place for a long time, all replicas will gradually become 100% consistent.
  - This model guarantees that updates to be propagated to all replicas eventually.
  - Write-write conflicts are often relatively easy to solve assuming only a small group of processes
  - can perform updates.
  - Very inexpensive to implement.

- ✓ **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 or a more recent value
  - This guarantees that if a process has seen a value x at time t, it will never see an older version of x afterwards.
  - Example: Distributed e-mail database
    - Updates are propagated in a lazy fashion
    - Reading (not modifying) incoming e-mails 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 connected

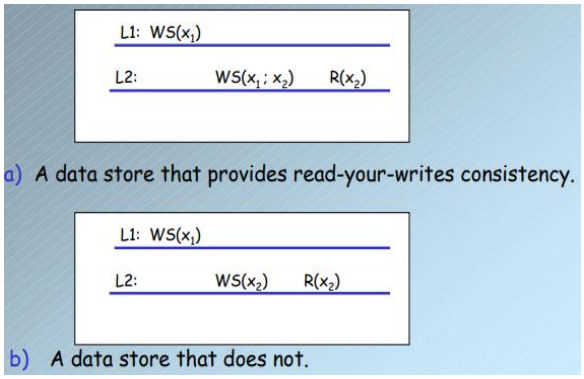


✓ **Monotonic Writes**

- A write operation by a process on a data item x is completed before any successive write operation on x by the same process
  - Resembles the FIFO consistency; different in the sense that monotonic writes model cares what the single process, which is performing writes, sees.
  - A write operation on a copy of data item x is performed only if that copy has been brought up to date by means of any preceding write operation that may have taken place on other copies of x by the same process.
  - Example: Updating a C library at server Si and ensuring that all previous updates on all components of the library on which compilation and linking depends, are also propagated to Si



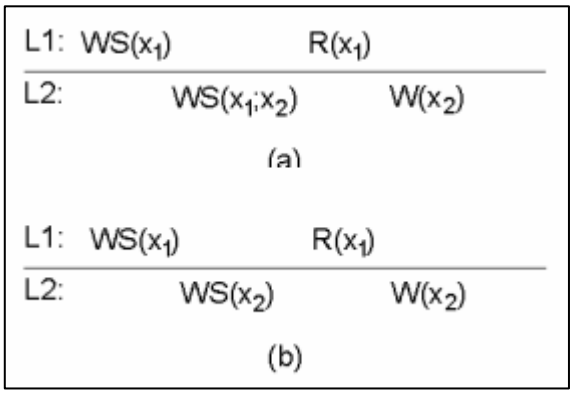
✓ **Read Your Writes**



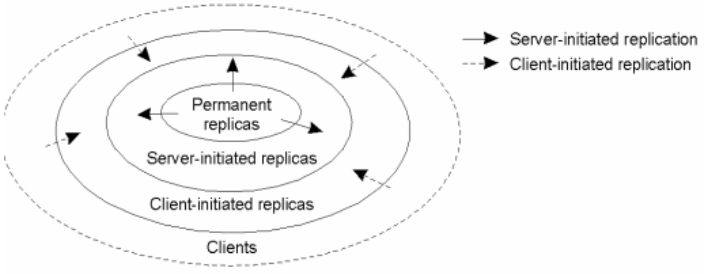
- The effect of a write operation by a process on data item x, will always be seen by a successive read operation on x by the same process
- A write operation is always completed before a successive read operation by the same process, no matter where this read operation takes place.
- Example: Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.
- Example: updating passwords on password server.

✓ **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.
- Any successive write operation by a process on x will be performed on a copy of x that' s up to date with the value most recently read by that process
- Example:
  - a) A writes-follow-reads consistent data store
  - b) A data store that does not provide writes-follow-reads consistency



4. Content Replication and placement (Permanent Replica, Server initiated, Client initiated)



The logical organization of different kinds of copies of a data store into three concentric rings.

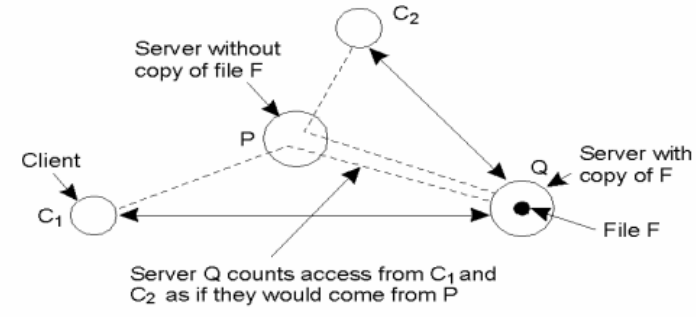
✓ We need to decide where, when and by whom copies of the data-store are to be placed.
✓ Three types of copies are Permanent replicas, Server-initiated replicas, Client-initiated replicas

✓ **Permanent replicas:**

- Permanent replicas can be considered as the initial set of replicas that constitute a distributed data store. In many cases, the number of permanent replicas is small.
- Consider, for example, a Web site. Distribution of a Web site generally comes in one of two forms.
  - The first kind of distribution is one in which the files that constitute a site are replicated across a limited number of servers at a single location. Whenever a request comes in, it is forwarded to one of the servers, for instance, using a round-robin strategy.
  - The second form of distributed Web sites is what is called mirroring. In this case, a Web site is copied to a limited number of servers, called mirror sites which are geographically spread across the Internet. In most cases, clients simply choose one of the various mirror sites from a list offered to them. Mirrored Web sites have in common with cluster-based Web sites that there are only a few number of replicas, which are more or less statically configured.

✓ **Server-Initiated Replicas**

- In contrast to permanent replicas, server-initiated replicas are copies of a data store that exist to enhance performance and which are created at the initiative of the (owner of the) data store.
- Consider, for example, a Web server placed in New York. Normally, this server can handle incoming requests quite easily, but it may happen that over a couple of days a sudden burst of requests come in from an unexpected location far from the server. In that case, it may be worthwhile to install a number of temporary replicas in regions where requests are coming from.
- To provide optimal facilities such hosting services can dynamically replicate files to servers where those files are needed to enhance performance, that is, close to demanding (groups of) clients.
- The algorithm for dynamic replication takes two issues into account.
  - First, replication can take place to reduce the load on a server.
  - Second, specific files on a server can be migrated or replicated to servers placed in the proximity of clients that issue many requests for those files.



Counting access requests from different clients.

C_1 and C_2 share the same closest server (P). Then, all accesses for file F at Q from C_1 and C_2 are jointly counted as $CNT_Q(P, F)$

✓ **Client-initiated Replicas**

- An important kind of replica is the one initiated by a client. Client-initiated replicas are more commonly known as (client) caches.
- In essence, a cache is a local storage facility that is used by a client to temporarily store a copy of the data it has just requested.
- In principle, managing the cache is left entirely to the client.
- The data store from where the data had been fetched has nothing to do with keeping cached data consistent. However, as we shall see, there are many occasions in which the client can rely on participation from the data store to inform it when cached data has become stale.
- Client caches are used only to improve access times to data.

Summary

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

Chap 6

1. **Explain Distributed File Systems (DFS)**

✓ A Distributed File System (DFS) as the name suggests, is a file system that is distributed on multiple file servers or multiple locations.

✓ It allows programs to access or store isolated files as they do with the local ones, allowing programmers to access files from any network or computer.

✓ The main purpose of the Distributed File System (DFS) is to allows users of physically distributed systems to share their data and resources by using a Common File System.

✓ Distributed file system support:

- **Remote Information Sharing** - Allows a file to be transparently accessed by processes of any node of the system irrespective of the file's location.
- **User Mobility** - User has flexibility to work on different nodes at different times.
- **Availability** - Better fault tolerance.
- **Diskless Workstations**

✓ Features of DFS

- **Transparency**:
  - *Structure Transparency:* Clients don't need to know the number or location of file servers. Multiple servers improve performance and reliability.
  - *Access Transparency:* Local and remote files are accessed the same way; the system automatically locates and sends files to the client.
  - *Naming Transparency:* File names don't reveal their location and remain unchanged when moved between nodes.
  - *Replication Transparency:* Copies of files on different nodes are hidden, ensuring seamless access.
- **User mobility:** It will automatically bring the user's home directory to the node where the user logs in.
- **Performance:** Performance is based on the average amount of time needed to convince the client requests. This time covers the CPU time + time taken to access secondary storage + network access time. It is advisable that the performance of the Distributed File System be similar to that of a centralized file system.
- **Simplicity and ease of use:** The user interface of a file system should be simple and the number of commands in the file should be small.
- **High availability:** A Distributed File System should be able to continue in case of any partial failures like a link failure, a node failure, or a storage drive crash.
- **High reliability:** Probability of loss of stored data should be minimized. System should automatically generate backup copies of critical files.
- **Scalability:** Growth of nodes and users should not seriously disrupt service
- **Data integrity:** Concurrent access requests from multiple users who are competing to access the file must be properly synchronized by the use of some form of concurrency control mechanism.
- **Security:** Users should be confident of the privacy of their data.
- **Heterogeneity:** There should be easy access to shared data on diverse platforms.

2. File-Caching Schemes

A file-caching scheme for a distributed file system contributes to its scalability and reliability as it is possible to cache remotely located data on a client node. Every distributed file system use some form of file caching.

The following can be used:

➢ Cache Location

Cache location is the place where the cached data is stored. There can be three possible cache locations

- **Servers main memory:**

A cache located in the server's main memory eliminates the disk access cost on a cache hit which increases performance compared to no caching.

The reason for keeping locating cache in server's main memory-

Easy to implement

Totally transparent to clients

Easy to keep the original file and the cached data consistent.

- **Clients disk:**

If cache is located in clients disk it eliminates network access cost but requires disk access cost on a cache hit. This is slower than having the cache in servers main memory. Having the cache in server's main memory is also simpler.

- **Advantages:**

Provides reliability.

Large storage capacity.

Contributes to scalability and reliability.

- **Disadvantages:**

Does not work if the system is to support diskless workstations.

Access time is considerably large

➢ Clients main memory

A cache located in a client's main memory eliminates both network access cost and disk access cost. This technique is not preferred to a client's disk cache when large cache size and increased reliability of cached data are desired.

- **Advantages:**

Maximum performance gain.

Permits workstations to be diskless.

Contributes to reliability and scalability.

➢ Modification propagation

When the cache is located on client's nodes, a files data may simultaneously be cached on multiple nodes. It is possible for caches to become inconsistent when the file data is changed by one of the clients and the corresponding data cached at other nodes are not changed or discarded.

The modification propagation scheme used has a critical effect on the systems performance and reliability.

Techniques used include –

- **Write-through scheme**

When a cache entry is modified, the new value is immediately sent to the server for updating the master copy of the file.

- **Advantage:**

High degree of reliability and suitability for UNIX-like semantics.

The risk of updated data getting lost in the event of a client crash is low.

- **Disadvantage:**

Poor Write performance.

- **Delayed-write scheme**

To reduce network traffic for writes the delayed-write scheme is used. New data value is only written to the cache when a entry is modified and all updated cache entries are sent to the server at a later time.

There are three commonly used delayed-write approaches:

Write on ejection from cache:

Modified data in cache is sent to server only when the cache-replacement policy has decided to eject it from client's cache. This can result in good performance but there can be a reliability problem since some server data may be outdated for a long time.

Periodic write:

The cache is scanned periodically and any cached data that has been modified since the last scan is sent to the server.

Write on close:

Modification to cached data is sent to the server when the client closes the file. This does not help much in reducing network traffic for those files that are open for very short periods or are rarely modified.

Advantages:

Write accesses complete more quickly that result in a performance gain.

Disadvantage:

Reliability can be a problem.

➢ Cache validation schemes

The modification propagation policy only specifies when the master copy of a file on the server node is updated upon modification of a cache entry. It does not tell anything about when the file data residing in the cache of other nodes is updated. A file data may simultaneously reside in the cache of multiple nodes. A client's cache entry becomes stale as soon as some other client modifies the data corresponding to the cache entry in the master copy of the file on the server. It becomes necessary to verify if the data cached at a client node is consistent with the master copy. If not, the cached data must be invalidated and the updated version of the data must be fetched again from the server.

There are two approaches to verify the validity of cached data:

- **Client-initiated approach**

The client contacts the server and checks whether its locally cached data is consistent with the master copy.

Checking before every access- This defeats the purpose of caching because the server needs to be contacted on every access.

Periodic checking- A check is initiated every fixed interval of time.

Server-initiated approach

A client informs the file server when opening a file, indicating whether a file is being opened for reading, writing, or both. The file server keeps a record of which client has which file open and in what mode. The server monitors file usage modes being used by different clients and reacts whenever it detects a potential for inconsistency. E.g. if a file is open for reading, other clients may be allowed to open it for reading, but opening it for writing cannot be allowed. So also, a new client cannot open a file in any mode if the file is open for writing.

When a client closes a file, it sends intimation to the server along with any modifications made to the file. Then the server updates its record of which client has which file open in which mode.

When a new client makes a request to open an already open file and if the server finds that the new open mode conflicts with the already open mode, the server can deny the request, queue the request, or disable caching by asking all clients having the file open to remove that file from their caches.

3. File Accessing models

- ✓ File Accessing Models depends on the following two factors:
  - 1. Method of Accessing remote files
  - 2. Unit of Data Transfer
- ✓ Based on the **unit of data access**, following file access models may be utilized to get to the particular file.
  1. **File-level transfer model:** In file level transfer model, the all-out document is moved while a particular action requires the document information to be sent the whole way through the circulated registering network among client and server. This model has better versatility and is proficient.
  2. **Block-level transfer model:** In the block-level transfer model, record information travels through the association among client and a server is accomplished in units of document blocks. Thus, the unit of information move in block-level transfer model is document blocks. The block-level transfer model might be used in dispersed figuring climate containing a few diskless workstations.
  3. **Byte-level transfer model:** In the byte-level transfer model, record information moves the association among client and a server is accomplished in units of bytes. In this way, the unit of information move in byte-level exchange model is bytes. The byte-level exchange model offers more noteworthy versatility in contrast with the other record move models since, it licenses recuperation and limit of a conflicting progressive sub range of a document. The significant hindrance to the byte-level exchange model is the trouble in store organization because of the variable-length information for different access requests.
  4. **Record-level transfer model:** The record-level file transfer model might be used in the document models where the document contents are organized as records. In record-level exchange model, document information travels through the organization among client and a server is accomplished in units of records. The unit of information move in record-level transfer model is record.
- ✓ Based on the **Method Utilizes for Accessing Remote Files:** A distributed file system might utilize one of the following models to service a client's file access request when the accessed to file is remote:
  1. **Remote service model:** Handling of a client's request is performed at the server's hub. Thusly, the client's solicitation for record access is passed across the organization as a message on to the server, the server machine plays out the entrance demand, and the result is shipped off the client. Need to restrict the number of messages sent and the vertical per message.
     - Remote access is taken care of across the organization so it is all the slower.
     - Increase server weight and organization traffic. Execution undermined.
     - Transmission of series of responses to explicit solicitation prompts higher organization overhead.
     - For staying aware of consistency correspondence among client and server is there to have a specialist copy predictable with clients put away data.
     - Remote assistance better when essential memory is close to nothing.
     - It is only an augmentation of neighbourhood record system interface across the network.
  2. **Data-caching model:** This model attempts to decrease the organization traffic of the past model by getting the data got from the server center. This exploits the region part of the found in record gets to. A replacement methodology, for instance, LRU is used to keep the store size restricted.
     - Remote access can be served locally so that access can be quicker.
     - Network traffic, server load is reduced. Further develops versatility.
     - Network over head is less when transmission of huge of information in comparison to remote service.
     - For keeping up with consistency, if less writes than better performance in maintaining consistency, if more frequent writes, then poor performance.
     - Caching is better for machines with disk or large main memory.
     - Lower-level machine interface is different from upper-level UI (user interface).
     - **Benefit of Data-caching model over the Remote service model:**
     - The data -catching model offers the opportunity for expanded execution and greater system versatility since it diminishes network traffic, conflict for the network, and conflict for the document servers. Hence almost all distributed file systems implement some form of caching.
     - **Example:** NFS utilizes the remote service model but adds caching for better execution.