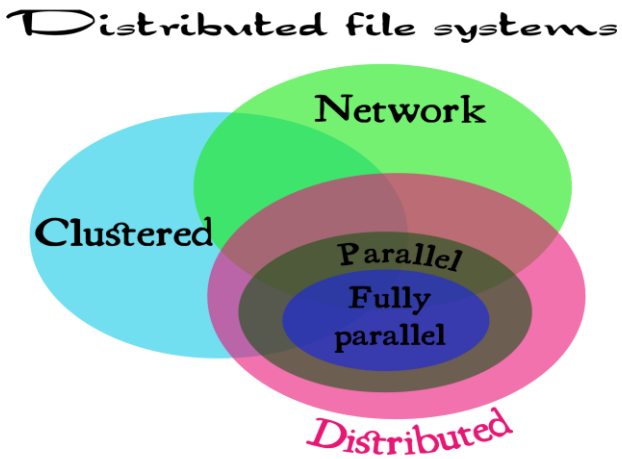
# MODULE-5,6: Consistency and Distributed file systems

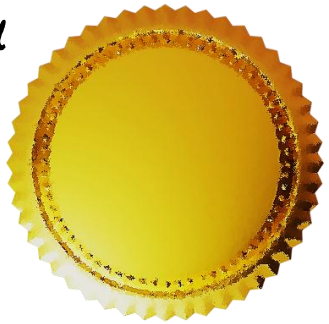
VIT | Vidyalankar Institute of Technology
Accredited A+ by NAAC




**Prepared by Prof. Amit K. Nerurkar**

# Certificate

This is to certify that the e-book titled "CONSISTENCY AND DISTRIBUTED FILE SYSTEMS" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.



Signature Date: 20-03-2020

Prof. Amit K. Nerurkar
Assistant Professor
Department of Computer Engineering

⚠ *DISCLAIMER: The information contained in this e-book is compiled and distributed for educational purposes only. This e-book has been designed to help learners understand relevant concepts with a more dynamic interface. The compiler of this e-book and Vidyalankar Institute of Technology give full and due credit to the authors of the contents, developers and all websites from wherever information has been sourced. We acknowledge our gratitude towards the websites YouTube, Wikipedia, and Google search engine. No commercial benefits are being drawn from this project.*

# Module 5 & 6 Consistency and Replication & Distributed File Systems

## 5 CONSISTENCY AND REPLICATION

### INTRODUCTION

Special attention is paid to replicating objects, as this forms an increasingly important topic in modern distributed systems.

**Reasons for Replication**

There are two primary reasons for replicating data: reliability and performance. First, data are replicated to increase the reliability of a system. If a file system has been replicated it may be possible to continue working after one replica crashes by simply switching to one of the other replicas. Also, by maintaining multiple copies, it becomes possible to provide better protection against corrupted data. For example, imagine there are three copies of a file, and every read and write operation is performed on each copy. We can safeguard ourselves against a single, failing write operation, by considering the value that is returned by at least two copies as being the correct one.

The other reason for replicating data is performance. Replication for performance is important when the distributed system needs to scale in numbers and geographical area. Scaling in numbers occurs, for example, when an increasing number of processes needs to access data that are managed by a single server. In that case, performance can be improved by replicating the server and subsequently dividing the work.

### DATA-CENTRIC CONSISTENCY MODELS

**Q.1 Explain Data-Centric Consistency Models.**

**(A)** Traditionally, consistency has always been discussed in the context of read and write operations on shared data, available by means of (distributed) shared memory, a (distributed) shared database, or a (distributed) file system. A data store may be physically distributed across multiple machines. In particular, each process that can access data from the store is assumed to have a local (or nearby) copy available of the entire store. Write operations are propagated to the other copies, as shown in Fig.1. A data operation is classified as a write operation when it changes the data, and is otherwise classified as a read operation.

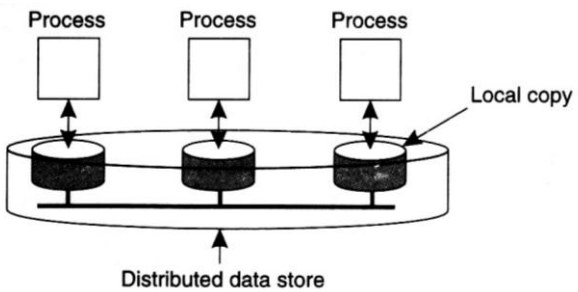



**Fig.1 :** The general organization of a logical data store, physically distributed and replicated across multiple processes.

A **consistency model** is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly. Normally, a process that performs a read operation on a data item, expects the operation to return a value that shows the results of the last write operation on that data.

**Strict Consistency**

The most stringent consistency model is called strict consistency. It is defined by the following condition:

*Any read on a data item x returns a value corresponding to the result of the most recent write on x.*

This definition is natural and obvious, although it implicitly assumes the existence of absolute global time so that the determination of "most recent" is unambiguous. Uniprocessor systems have traditionally observed strict consistency and uniprocessor programmers have come to expect such behavior as a matter of course. A system on which the program

a = 1; a = 2; print(a);

printed 1 or any value other than 2 would quickly lead to a lot of very agitated programmers, and for good reason too.

The problem with strict consistency is that it relies on absolute global time. In essence, it is impossible in a distributed system to assign a unique timestamp to each operation that corresponds to actual global time. We can relax this situation by dividing time into a series of consecutive, non-overlapping intervals. Each operation is assumed to take place within an interval and receives a timestamp, that corresponds to that interval. Depending on how accurate clocks can be synchronized, we may now reach a situation in which there is at most one operatic per interval.


**Fig.2 :** Behavior of two processes operating on the same data item. The horizontal axis is time. (a) A strictly consistent store. (b) A store that is not strictly consistent

As an example, in Fig. 2(a) below $P_1$ does a write to a data item x, modifying its value to a. Note that, in principle, this operation $W_1(x)a$ is first performed on a copy of the data store that is local to $P_1$ , and is then subsequently propagated to the other local copies. In our example, P2 later reads x (from its local copy of the store and sees value a. This behavior is correct for a strictly consistent data store. In contrast, in Fig. 2(b), $P_2$ does a read after the write (possibly only a nanosecond after it, but still after it), and gets NIL. A subsequent read returns a. Such behavior is incorrect for a strictly consistent data store.

**Linearizability and Sequential Consistency**

While strict consistency is the ideal consistency model, it is impossible to implement in a distributed system. Furthermore, experience shows that programmers can often manage quite well with weaker models. Counting on two events within one process happening so quickly that the other process will not be able to do something in between is looking for trouble. Instead, the reader is taught to program in such a way that the exact order of statement execution (in fact, memory references) does not matter. When the order of events is essential, semaphores or other synchronization operations should be used. Accepting this argument means learning to live with a weaker consistency model.

**Sequential consistency** is a slightly weaker consistency model than strict consistency. It was first defined by Lamport, in the context of shared memory for multiprocessor systems. In general, a data store is said to be sequentially consistent when it satisfies the following condition:

*The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.*

What this definition means is that when processes run concurrently on (possibly) different machines, any valid interleaving of read and write operations is acceptable behavior, but all processes see the same interleaving of operations. Note that nothing is said about time; that is, there is no reference to the "most recent" write operation on an object. Note that in this context, a process "sees" writes from all processes but only its own reads.

That time does not play a role can be seen from Fig.3. Consider four processes operating on the same data item x. In Fig. 3(a) process $P_1$ first performs W(x)a to x. Later (in absolute time), process P2 also performs a write operation, by setting the value of x to b. However, both processes P3 and P4 first read value b, and later value a. In other words, the write operation of process P2 appears to have taken place before that of $P_1$.

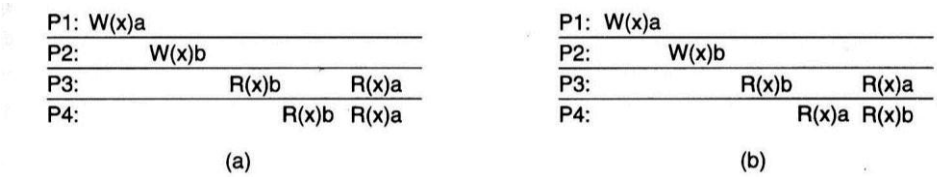



**Fig. 3 :** (a) A sequentially consistent data store. (b) A data store that is not sequentially consistent.

In contrast, Fig.3(b) violates sequential consistency because not all processes see the same interleaving of write operations. In particular, to process P3, it appears as if the data item has first been changed to b, and later to a. On the other hand, P4 will conclude that the final value is b.

A consistency model that is weaker than strict consistency, but stronger than sequential consistency, is linearizability. In this model, operations are assumed to receive a timestamp using a globally available clock, but one with only finite pre-

cision. Such a clock can be implemented in a distributed system by assuming processes use loosely synchronized clocks Let tsop(x) denote the timestamp assigned to operation OP that is performed on data item x, where OP is either a read (R) or write (W). A data store is said to be linearizable when each operation is timestamped and the following condition holds:

---

*The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program. In addition, if* $tS_{OP1}(x) < tS_{OP2}(y)$ *, then operation OP l(x) should precede OP2(y) in this sequence.*

---

**Causal Consistency**

The **causal consistency** model represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not. We already came across causality when discussing vector timestamps in the previous chapter. If event B is caused or influenced by an earlier event, A, causality requires that everyone else first see A, then see B.

Consider a memory example. Suppose that process $P_1$ writes a variable x. Then P2 reads x and writes y. Here the reading of x and the writing of y are potentially causally related because the computation of y may have depended on the value of x read by P2 (i.e., the value written by $P_1$). On the other hand, if two processes spontaneously and simultaneously write two different variables, these are not causally related. When there is a read followed later by a write, the two events are potentially causally related. Similarly, a read is causally related to the write that provided the data the read got. Operations that are not causally related are said to be concurrent.

For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

---

*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.*

---

As an example of causal consistency, consider Fig.4. Here we have an event sequence that is allowed with a causally-consistent store, but which is forbidden with a sequentially consistent store or a strictly consistent store. The thing to note is that the writes W2(x)b and W\(x)c are concurrent, so it is not required that all processes see them in the same order.

| | | | | | |
|---|---|---|---|---|---|
| P1: | W(x)a | | W(x)c | | |
| P2: | | R(x)a | W(x)b | | |
| P3: | | R(x)a | | R(x)c | R(x)b |
| P4: | | R(x)a | | R(x)b | R(x)c |

**Fig. 4 :** This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

Now consider a second example. In Fig. 5 we have $W_2(x)b$ potentially depending on $W_1(x)$ a because the b may be a result of a computation involving the value read by $R_2(x)a$. The two writes are causally related, so all processes must see them in the same order. Therefore, Fig.5(a) is incorrect. On the other hand, in Fig.5(b) below the read has been removed, so $W_1(x)a$ and $W_2(x)b$ are now concurrent writes. A causally-consistent store does not require concurrent writes to be globally ordered, so Fig. 5(b) is correct.

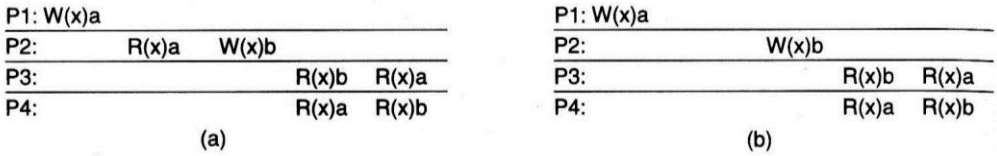



**Fig.5 :** (a) A violation of a causally consistent store.
(b) A correct sequence of events in a causally consistent store.

Implementing causal consistency requires keeping track of which processes have seen which writes. It effectively means that a dependency graph of which operation is dependent on which other operations must be constructed and maintained.

**FIFO Consistency**

In causal consistency, it is permitted that concurrent writes be seen in a different order on different machines, although causally-related ones must be seen in the same order by all machines. The next step in relaxing consistency is to drop the latter requirement. Doing so gives FIFO consistency, which is subject to the 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.*

FIFO consistency is called PRAM consistency in the case of distributed shared memory systems and is described in. PRAM stands for Pipelined RAM, because writes by a single process can be pipelined, that is, the process does not have to stall waiting for each one to complete before starting the next one. FIFO consistency is contrasted with causal consistency in Fig.6. The sequence of-events shown here is allowed with a FIFO consistent data store but not with any of the stronger models we have studied so far.

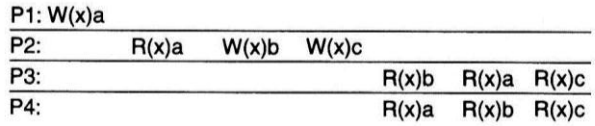



**Fig.6 :** A valid sequence of events for FIFO consistency.

FIFO consistency is interesting because it is easy to implement. In effect it says that there are no guarantees about the order in which different processes see writes, except that two or more writes from a single source must arrive in order. Put in other terms, in this model all writes generated by different processes are concurrent. The model can be implemented by simply tagging each write opera-

tion with a (process, sequence number) pair, and performing writes per process in the order of their sequence number.

**Weak Consistency**

Although FIFO consistency can give better performance than the stronger consistency models, it is still unnecessarily restrictive for many applications because they require that writes originating in a single process be seen everywhere in order. Consider the case of a process inside a critical section writing records to a replicated database. Even though other processes are not supposed to touch the records until the first process has left its critical section, the database system has no way of knowing when a process is in a critical section and when it is not, so it has to propagate all writes to .all copies of the database.

A better solution would be to let the process finish its critical section and then make sure that the final results are sent everywhere, not worrying too much whether all intermediate results have also been propagated to all copies in order, or even at all. In general, this can be done by introducing what is called a synchronization variable. A synchronization variable 5 has only a single associated operation synchronize(S), which synchronizes all local copies of the data store. Recall that a process P performs operations only on its locally available copy of the store. When the data store is synchronized, all local writes by process P are propagated to the other copies, whereas writes by other processes are brought in to P's copy.

Using synchronization variables to partly define consistency leads to what is called weak consistency. Weak consistency models have three properties:

1. *Accesses to synchronization variables associated with a data store, are sequentially consistent.*
2. *No operation on a synchronization variable is allowed to be performed until all previous writes have completed everywhere.*
3. *No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.*

Now let us consider a somewhat less far-fetched situation. In Fig.7(a) we see that process $P_1$ does two writes to a data item, and then synchronizes (indicated by the letter S). If $P_2$ and $P_3$ have not yet been synchronized, no guarantees are given about what they see, so this sequence of events is valid.

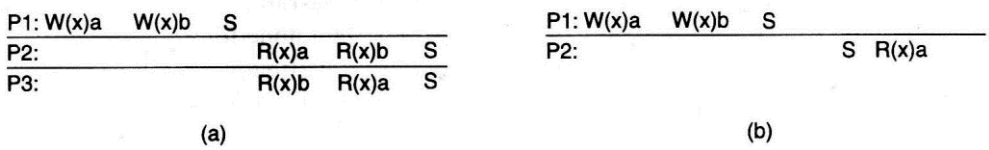



**Fig.7 :** (a) A valid sequence of events for weak consistency.
(b) An invalid sequence consistency.

Fig.7(b) is different. Here $P_2$ has been synchronized, which means that its local copy of the data stored is brought up to date. When it reads x, it must get the value b. Getting a, as shown in the figure, is not permitted with weak consistency.

**Release Consistency**

**Release consistency** provides these two kinds. An acquire operation is used to tell the data store that a critical region is about to be entered, whereas a release operation says that a critical region has just been exited. These operations can be implemented in either of two ways: (1) ordinary operations on special variables or (2) special operations. Either way, the programmer is responsible for inserting explicit code in the program stating when to do the operations, for example, by calling library procedures such as acquire and release or procedures such as enter_critical region and leave_critical_region.

In addition to these synchronizing operations, reading and writing shared data is also possible. Acquire and release do not have to apply to all data in a store Instead, they may guard only specific shared data, in which case only those data items are kept consistent. The shared data that are kept consistent are said to be **protected.**

Fig. 8 depicts a valid sequence of events for release consistency. Process $P_1$ does an acquire, changes a shared data item twice, and then does a release. Process $P_2$ does an acquire and reads data item x. It is guaranteed to get the value that x had at the time of the release, namely b (unless $P_2$'s acquire performs before $P_1$'s acquire). If the acquire had been done before $P_1$ did the release, the acquire would have been delayed until the release had occurred. Since $P_3$ does not do an acquire before reading shared data, the data store has no obligation to give it the current value of x, so returning a is allowed.

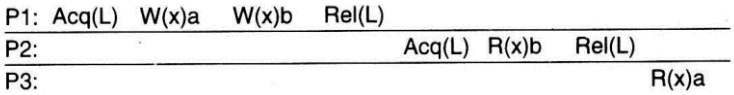



**Fig.8 :** A valid event sequence for release consistency.

While the centralized algorithm described above will do the job, it is by no means the only approach. In general, a distributed data store is release consistent if it obeys the following rules:

1. *Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed success-fully.*
2. *Before a release is allowed to be performed, all previous reads and writes done by the process must have been completed.*
3. *Accesses to synchronization variables are FIFO consistent (sequential consistency is not required*

**Entry Consistency**

Another consistency model that has been designed to be used with critical sections is entry consistency. Like both variants of release consistency, it requires the programmer (or compiler) to use acquire and release at the start and end of each critical section, respectively. However, unlike release consistency, entry consistency requires each ordinary shared data item to be associated with some synchronization variable such as a lock or barrier. If it is desired that

elements of an array be accessed independently in parallel, then different array elements must be associated with different locks.
Formally, a data store exhibits entry consistency if it meets all the following conditions :

1. *An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.*
2. *Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.*
3. *After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.*

Fig. 9 shows an example of entry consistency. Instead of operating on the entire shared data, in this example we associate locks with each data item. In this case, $P_1$ does an acquire for x, changes x once, after which it also does an acquire for y. Process $P_2$ does an acquire for x but not for y, so that it will read value a for x, but may read NIL for y. Because process $P_3$ first does an acquire for y, it will read the value b when y is released by $P_1$.

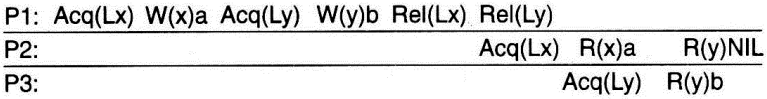



**Fig. 9 :** A valid event sequence for entry consistency.

**Summary of Consistency Models**

| Consistency | Description |
|---|---|
| Strict | Absolute time ordering of all shared accesses |
| Linearizability | All processes 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 issued. Writes from different processes may not always be seen in that order |

**(a)**

| 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 |

**(b)**

(a) Consistency models not using synchronization operations, (b) Models with synchronization operations.

In short, weak consistency, release consistency, and entry consistency require additional programming constructs that, when used as directed, allow programmers to pretend that a data store is sequentially consistent, when, in fact, it is not. In principle, these three models using explicit synchronization should be able to offer the best performance, but it is likely that different applications will give quite different results.

**Client-Centric Consistency Models**

**Q.2 Explain Client-Centric Consistency Models.**

**(A)** An important assumption is that concurrent processes may be simultaneously updating the data store, and that it is necessary to provide consistency in the face of such concurrency. For example, in the case of object-based entry consistency, the data store guarantees that when an object is invoked, the invoking process is provided with a copy of the object that reflects all changes to the object that have been made so far, possibly by other processes. During the invocation, it is also guaranteed that no other process can interfere, that is, mutual exclusive access is provided to the invoking process.

By introducing special client-centric consistency models, it turns out that many inconsistencies can be hidden in a relatively cheap way.

**Eventual Consistency**

To what extent processes actually operate in a concurrent fashion, and to what extent consistency needs to be guaranteed, may vary. There are many examples in which concurrency appears only in a restricted form. For example, in many database systems, most processes hardly ever perform update operations but only read data from the database. Only one, or very few processes perform update operations. The question then is how fast updates should be made available to only-reading processes.

As another example, consider a worldwide naming system such as DNS. The DNS name space is partitioned into domains, where each domain is assigned to a naming authority, which acts as owner of that domain. Only that authority is allowed to update its part of the name space. Consequently, conflicts resulting from two operations that both want to perform an update on the same data never occur (i.e., write-write conflicts). The only situation that needs to be handled are read-write conflicts. As it turns out, it is often acceptable to propagate an update in a lazy fashion, meaning that a reading process will see an update only after some time has passed since the update took place.

Yet another example is the World Wide Web. In virtually all cases, Web pages are updated by a single authority, such as a webmaster or the actual owner of the page. There are normally no write-write conflicts to resolve. On the other hand, to improve efficiency, browsers and Web proxies are often configured to keep a fetched page in a local cache, and to return that page upon the next

request. An important aspect of both types of Web caches is that they may return out-of-date Web pages. In other words, the cached page that is returned to the requesting client is an older version compared to the one available at the actual Web server. As it turns out, many users find this inconsistency acceptable.

These examples can be viewed as cases of (large-scale) distributed and replicated databases that tolerate a relatively high degree of inconsistency. They have in common that if no updates take place for a long time, all replicas will gradually become consistent. This form of consistency is called eventual consistency.

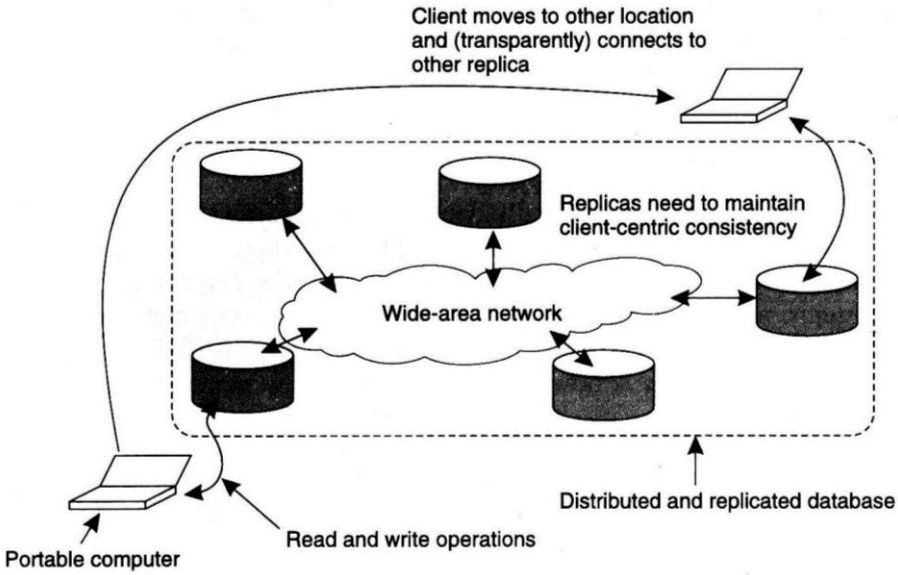



**Fig.1 :** The principle of a mobile user accessing different replicas of a distributed database.

Eventual consistent data stores work fine as long as clients always access the same replica. However, problems arise when different replicas are accessed. This is best illustrated by considering a mobile user accessing a distributed database as shown in Fig. 1.

The mobile user accesses the database by connecting to one of the replicas in a transparent way. In other words, the application running on the user's portable computer is unaware on which replica it is actually operating. Assume the user performs several update operations and then disconnects again. Later, he accesses the database again, possibly after moving to a different location or by using a different access device. At that point, the user may be connected to a different replica than before, as shown in Fig.1. However, if the updates performed previously have not yet been propagated, the user will notice inconsistent behavior. In particular, he would expect to see all previously-made changes, but instead, it appears as if nothing at all has happened.

**Monotonic Reads**

The first client-centric consistency model is that of monotonic reads. A data store is said to provide monotonic-read consistency if the following condition holds:

*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.*

In other words, monotonic-read consistency guarantees that if a process has seen a value of x at time t, it will never see an older version of x at a later time.
Using a notation similar to that for data-centric consistency models, monotonic-read consistency can be graphically represented as shown in Fig. 2. Along the vertical axis, two different local copies of the data store are shown, L1 and L2. Time is shown along the horizontal axis.

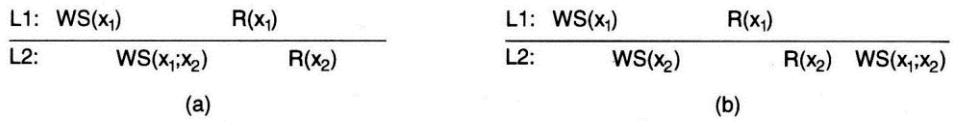



**Fig.2 :** The read operations performed by a single process P at two different local copies of the same data store. (a) A monotonic-read consistent data store. (b) A data store that does not provide monotonic reads.

In Fig.2(a), process P first performs a read operation on x at $L_1$, returning the value of $x_1$ (at that time). This value results from the write operations in WS ($x_1$) performed at $L_1$ . Later, P performs a read operation on x at $L_2$, shown as $R(x_2)$. To guarantee monotonic-read consistency, all operations in $WS(x_1)$ should have been propagated to $L_2$ before the second read operation takes place. In other words, we need to know for sure that $WS(x_1)$ is part of $WS(x_2)$, which is expressed as $WS(x_1 ; x_2)$.

In contrast, Fig.2(b) shows a situation in which monotonic-read consistency is not guaranteed. After process P has read $x_1$ at $L_1$, it later performs the operation $R(x_2)$ at $L_2$. However, only the write operations in $WS(x_2)$ have been performed at $L_2$. No guarantees are given that this set also contains all operations contained in WS ($x_1$).

**Monotonic Writes**
In many situations, it is important that write operations are propagated in the correct order to all copies of the data store. This property is expressed in monotonic-write consistency. In a **monotonic-write consistent** store, the following condition holds:

*A write operation by a process on a data item x is completed before any successive write operation on x by the same process.*

Thus completing a write operation means that the copy on which a successive operation is performed, reflects the effect of a previous write operation by the same process, no matter where that operation was initiated. In other words, 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, which may have taken place on other copies of x.

Monotonic-write consistency is shown in Fig. 3. In Fig.3(a), process P performs a write operation on x at local copy $L_1$, presented as the operation $W(x_1)$. Later, P performs another write operation on x, but this time at $L_2$, shown as $W(x_2)$. To ensure monotonic-write consistency, it is necessary that the previous write operation at $L_1$ has already been propagated to $L_2$. This explains operation $W(x_1)$ at $L_2$ and why it takes place before $W(x_2)$.


**Fig. 3 :** The write operations performed by a single process P at two different local copies of the same data store. (a) A monotonic write consistent data store. (b) A data store that does not provide monotonic-write consistency.

In contrast, Fig.3(b) shows a situation in which monotonic-write consistency is not guaranteed. Compared to Fig.3(a), what is missing is the propagation of $W(x_1)$ to copy $L_2$. In other words, no guarantees can be given that the copy of x on which the second write is being performed, has the same or more recent value at the time $W(x_1)$ completed at $L_1$.

**Read Your Writes**

A client-centric consistency model that is closely related to monotonic reads, is as follows. A data store is said to provide read-your-writes consistency, if the following condition holds:

*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.*

In other words, a write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.

The absence of read-your-writes consistency is often experienced when updating Web HTML pages and subsequently viewing the effects. Update operations often take place by means of a standard editor or word processor, which saves the new version on a file system that is shared by the Web server. The user's Web browser accesses that same file, possibly after requesting it from the local Web server. However, once the file has been fetched, either the server or the browser often caches a local copy for subsequent accesses. Consequently, when the Web page is updated, the user will not see the effects if the browser or the server returns the cached copy instead of the original file. Read-your-writes consistency can guarantee that if the editor and browser are integrated into a single program, the cache is invalidated when the page is updated, so that the updated file is fetched and displayed.

Fig.4(a) shows a data store that provides read–your writes consistency.

In Fig.4(a), process P performed a write operation $W(x_1)$ and later a read operation at a different local copy. Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation.

This is expressed by $WS(x_1;x_2)$, which states that $W(x_1)$ is part of $WS(x_2)$. In contrast, in Fig.4(b), $W(x_1)$ has been left out of $WS(x_2)$, meaning that the effects of the previous write operation by process P have not been propagated to $L_2$.

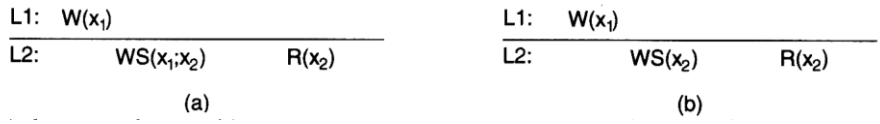



**Fig. 4 :** (a) A data store that provides read-your-writes consistency. (b) A data store that does not.

**Writes Follow Reads**

The last client-centric consistency model is one in which updates are propagated as the result of previous read operations. A data store is said to provide **writes-follow-reads** consistency, if the following holds.

*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.*

In other words, any successive write operation by a process on a data item x will be performed on a copy of x that is up to date with the value most recently read by that process.

Writes-follow-reads consistency can be used to guarantee that users of a net-work newsgroup see a posting of a reaction to an article only after they have seen the original article. To understand the problem, assume that a user first reads an article A. Then, he reacts by posting a response B. By requiring writes-follow-reads consistency, B will be written to any copy of the newsgroup only after A has been written as well. Note that users who only read articles need not require any specific client-centric consistency model. The writes-follows-reads consistency assures that reactions to articles are stored at a local copy only if the original is stored there as well.


**Fig.5 :** (a) A writes-follow-reads consistent data store. (b) A data store that does not provide writes-follow-reads consistency.

This consistency is shown in Fig.5. In Fig.5(a), a process reads x at local copy $L_1$. The write operations that led to the value just read, also appear in the write set at $L_2$, where the same process later performs a write operation. (Note that other processes at $L_2$ see those write operations as well.) In contrast, no guarantees are given that the operation performed at $L_2$, as shown in Fig.5(b), are performed on a copy that is consistent with the one just read at $L_1$.

**Video**

**Q.3 Explain Consistency Protocols.**

**(A)** A consistency protocol describes an implementation of a specific consistency model. By-and-large, the consistency models in which operations are globally serialized are the most important and widely applied models. These models include sequential consistency, weak consistency with synchronization variables, as well as atomic transactions.

**Primary-Based Protocols**

In primary-based protocols, each data item x in the data store has an associated primary, which is responsible for coordinating write operations on x. A distinction can be made as to whether the primary is fixed at a remote server or if write operations can be carried out locally after moving the primary to the process where the write operation is initiated.

**Remote-Write Protocols**

The simplest primary-based protocol is the one in which all read and write operations are carried out at a (remote) single server. In effect, data are not replicated at all, but instead are placed at a single server from which they cannot be moved. This model is traditionally used in client-server systems, where the server possibly may be distributed. This protocol is shown in Fig. 1.

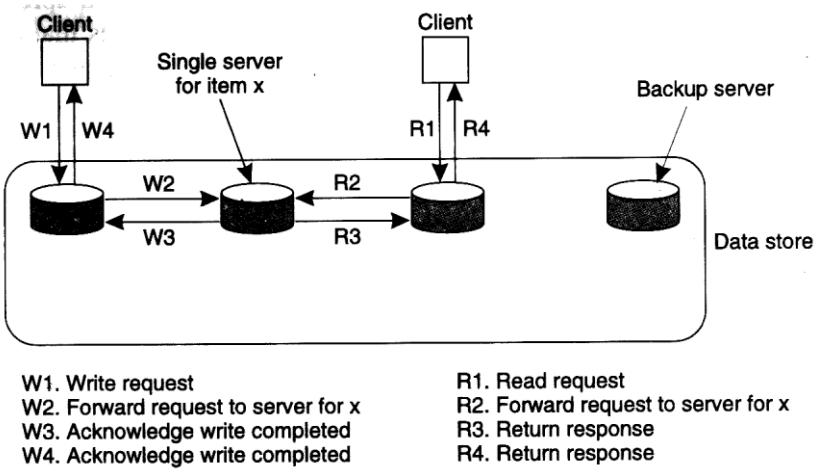



**Fig. 1 :** Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded.

More interesting from the perspective of consistency, are protocols that allow processes to perform read operations on a locally available copy, but should forward write operations to a (fixed) primary copy. Such schemes are frequently known as primary-backup protocols. A **primary-backup protocol** works as shown in Fig. 2. A process wanting to perform a write operation on data item x, forwards that operation to the primary server for x. The primary performs the update on its local copy of x, and subsequently forwards the update to the backup servers. Each backup server performs the update as well, and sends an acknowledgement back to the primary. When all backups have updated their local copy the primary sends an acknowledgement back to the initial process.

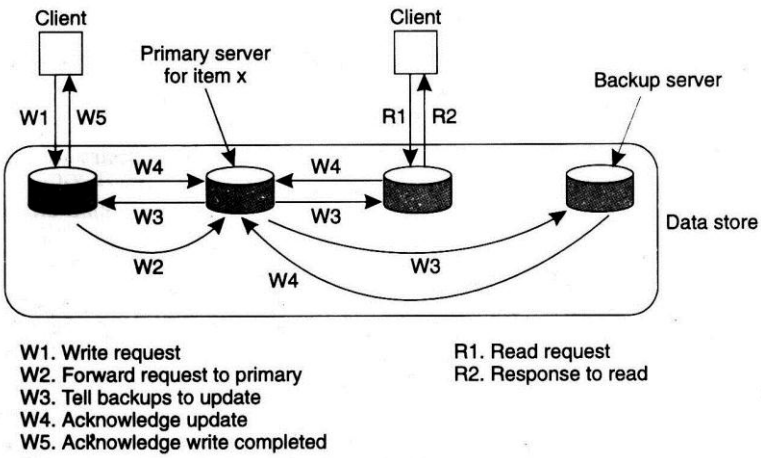



**Fig. 2 :** The principle of a primary–backup protocol.

Primary-backup protocols provide a straightforward implementation of sequential consistency, as the primary can order all incoming writes. Evidently, all processes see all write operations in the same order, no matter which backup server they use to perform read operations. Also, with blocking protocols, processes will always see the effects of their most recent write operation

**Local-Write Protocols**

There are two kinds of primary-based local-write protocols. The first kind is the one in which there is only a single copy of each data item x. In other words, there are no replicas. Whenever a process wants to perform an operation on some data item, the single copy of that data item is first transferred to the process, after which the operation is performed. This protocol essentially establishes a fully distributed nonreplicated version of the data store. Consistency is straightforward as there is always only a single copy of each data item. This protocol is shown in Fig. 3.

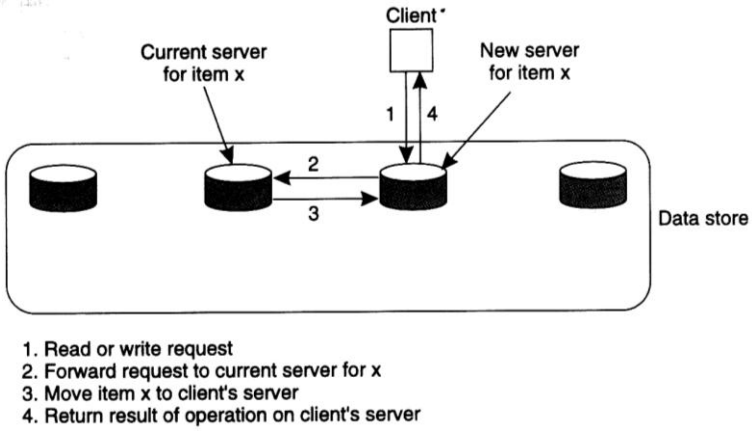



**Fig.3 :** Primary based local write protocol in which a single copy is migrated between processes.

One of the main problems with this fully migrating approach, is keeping track of where each data item currently is. Alternative solutions are the use of forwarding pointers and home-based approaches. Such solutions have been applied in distributed shared memory systems.

A variant of the local-write protocol just described is a primary-backup protocol in which the primary copy migrates between processes that wish to perform a write operation. As before, whenever a process wants to update data item x, it locates the primary copy of x, and subsequently moves it to its own location, as shown in Fig. 4. The main advantage of this approach is that multiple, successive write operations can be carried out locally, while reading processes can still access their local copy. However, such an improvement can be achieved only if a non-blocking protocol is followed by which updates are propagated to the replicas after the primary has received an update, as we explained above. This protocol has been applied to various distributed shared memory systems.

**Video**

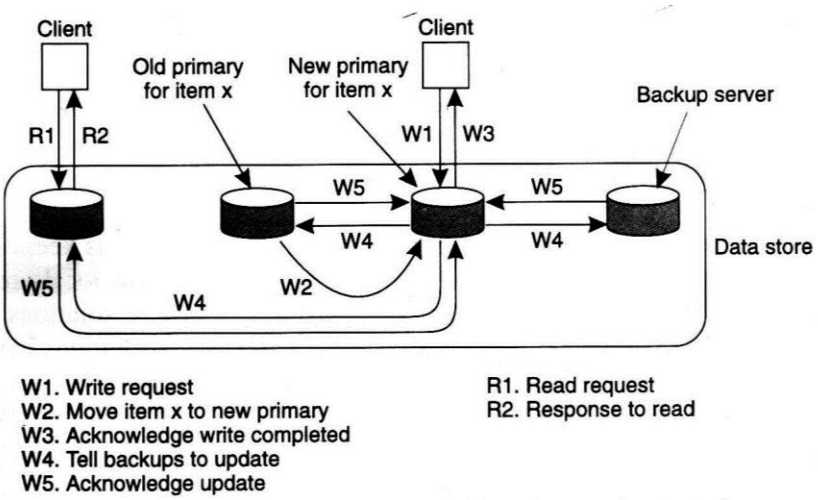



**Fig. 4 :** Primary backup protocol in which the primary migrates to the process wanting to perform an update.

**Replicated-Write Protocols**

In replicated-write protocols, write operations can be carried out at multiple replicas instead of only one, as in the case of primary-based replicas. A distinction can be made between active replication, in which an operation is forwarded to all replicas, and consistency protocols based on majority voting.

**Active Replication**

In active replication, each replica has an associated process that carries out update operations. In contrast to other protocols, updates are generally propagated by means of the write operation that causes the update. In other words, the operation is sent to each replica. However, it is also possible to send the update.

One potential problem with active replication is that operations need to be carried out in the same order everywhere. Consequently, what is needed is a totally-ordered multicast mechanism. Such a multicast can be implemented using Lamport timestamps, as discussed in the previous chapter. Unfortunately, using Lamport timestamps does not scale well in large distributed systems. As an alternative, total ordering can be achieved using a central coordinator, also called a sequencer. One approach is to first forward each operation to the sequencer, which assigns it a unique sequence number and subsequently forwards the operation to all replicas. Operations are carried out in the order of their sequence number. Clearly, this implementation of totally-ordered multicasting strongly resembles primary-based consistency protocols.

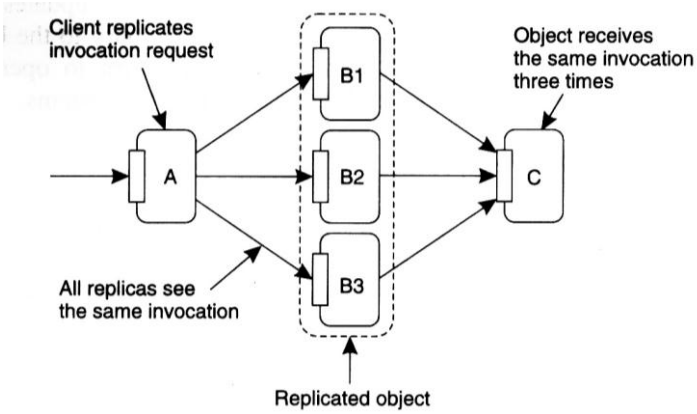



**Fig. 5 :** The problem of replicated invocations.

**Quorum-Based Protocols**

A different approach to supporting replicated writes is to use voting as originally proposed by Thomas and generalized by Gifford. The basic idea is to require clients to request and acquire the permission of multiple servers before either reading or writing a replicated data item.

As a simple example of how the algorithm works, consider a distributed file system and suppose that a file is replicated on N servers. We could make a rule stating that to update a file, a client must first contact at least half the servers plus one (a majority) and get them to agree to do the update. Once they have agreed, the file is changed and a new version number is associated with the new file. The version number is used to identify the version of the file and is the same for all the newly updated files.

To read a replicated file, a client must also contact at least half the servers plus one and ask them to send the version numbers associated with the file. If all the version numbers agree, this must be the most recent version because an attempt to update only the remaining servers would fail because there are not enough of them.

For example, if there are five servers and a client determines that three of them have version 8, it is impossible that the other two have version 9. After all, any successful update from version 8 to version 9 requires getting three servers to agree to it, not just two.

Gifford's scheme is actually somewhat more general than this. In it, to read a file of which N replicas exist, a client needs to assemble a **read quorum**, an arbitrary collection of any $N_R$ servers, or more. Similarly, to modify a file, a **write quorum** of at least $N_W$ servers is required. The values of $N_R$ and $N_W$ are subject to the following two constraints:

1. $N_R + N_W > N$
2. $N_W > N/2$

The first constraint is used to prevent read-write conflicts, whereas the second prevents write-write conflicts. Only after the appropriate number of servers has agreed to participate can a file be read or written.

Consider Fig.6(a), which has $N_R = 3$ and Nw = 10. Imagine that the most recent write quorum consisted of the 10 servers C through L. All of these get the new version and the new version number. Any subsequent read quorum of three servers will have to contain at least one member of this set. When the client looks at the version numbers, it will know which is most recent and take that one.

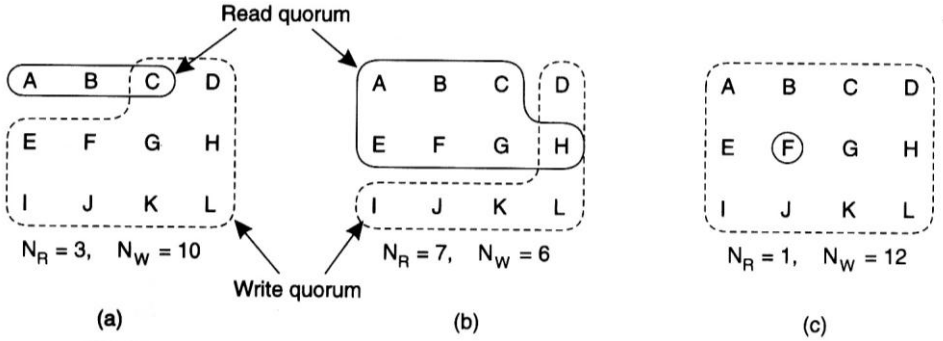



**Fig. 6 :** Three examples of the voting algorithm. (a) A correct choice of read and write set. (b) A choice that may lead to write-write conflicts. (c) A correct choice, known as ROWA (read one, write all)

In Fig. 6(b) and 6(c), we see two more examples. In Fig. 6(b) a write-write conflict may occur because $N_W \leq N/2$. In particular, if one client chooses {A,B,C,E,F,G} as its write set and another client chooses {D,H,I,J,K,L} as its write set, then clearly we will run into trouble as the two updates will both be accepted without detecting that they actually conflict.

The situation shown in Fig. 6(c) is especially interesting because it sets $N_R$ to one, making it possible to read a replicated file by finding any copy and using it. The price paid, however, is that write updates need to acquire all copies. This scheme is generally referred to as Read-One, Write-All (ROWA). There are several variations quorum-based replication protocols.

**Cache-Coherence Protocols**

Caches form a special case of replication, in the sense that they are generally controlled by clients instead of servers. However, cache-coherence protocols, which ensure that a cache is consistent with the server-initiated replicas are, in principle, not very different from the consistency protocols discussed so far.

There has been much research in the design and implementation of caches, especially in the context of shared-memory multiprocessor systems. Many solutions are based on support from the underlying hardware, for example, by assuming that snooping or efficient broadcasting can be done. In the context of middleware-based distributed systems that are built on top of general-purpose operating systems, software-based solutions to caches are more interesting.

Another design issue for cache-coherence protocols is the coherence enforcement strategy, which determines how caches are kept consistent with the copies stored at servers. The simplest solution is to disallow shared data to be cached at all. Instead, shared data are kept only at the servers, which maintain consistency using one of the primary-based or replication-write protocols discussed above. Clients are allowed to cache only private data. Obviously, this solution can offer only limited performance improvements.

When shared data can be cached, there are two approaches to enforce cache coherence. The first is to let a server send an invalidation to all caches whenever a data item is modified. The second is to simply propagate the update. Most caching systems use one of these two schemes. Dynamically choosing between sending invalidations or updates is sometimes supported in client-server databases.

Finally, we also need to consider what happens when a process modifies cached data. When read-only caches are used, update operations can be performed only by servers, which subsequently follow some distribution protocol to ensure that updates are propagated to caches. In many cases, a pull-based approach is followed. In this case, a client detects that its cache is stale, and requests a server for an update.

**Causally-Consistent Lazy Replication**

As a completely different example of consistency and replication, we now consider a replication scheme that implements eventual consistency, but at the same time keeps track of causal relationships between operations.

**System Model**

The essence of causally-consistent lazy replication, is that potential causality between read and write operations is captured by means of vector timestamps. We assume that the data store is distributed and replicated across N servers. As before, we assume that a client generally connects to the locally (or nearest) available server, although this is not strictly required.

Clients are allowed to communicate with each other, but will then have to exchange information on the operations they performed on the data store. The general organization of the data store is shown in Fig. 7.

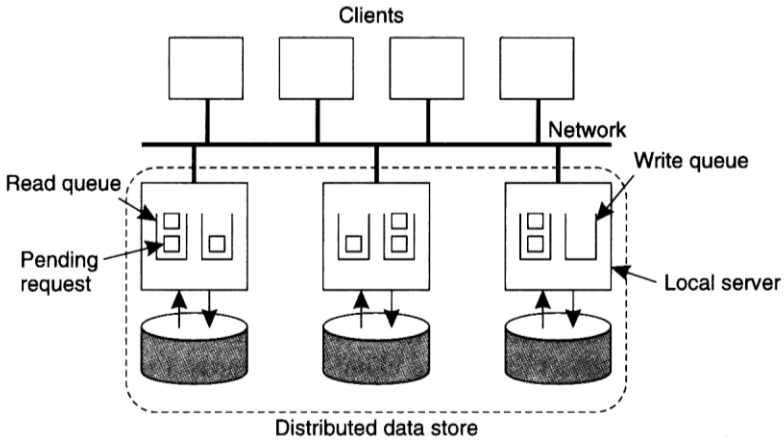



**Fig. 7 :** The general organization of a distributed data store.
Clients are assumed to also handle consistency related communication.

As is also shown in Fig. 7, each server of the data store consists of a local database and two queues of pending operations. The local database contains authoritative data, that is, data that could be permanently stored without violating the global consistency model of the data store. In other words, it contains precisely those data that match the software contract between the clients and the data store, expressed in the form of a data-centric consistency model. In this example, this contract enforces the local copies to match causal consistency.

# 6 Distributed File Systems

**INTRODUCTION**

In addition to the advantages of permanent storage and sharing of information provided by the file system of a single-processor system, a distributed file system normally supports the following:

i) Remote information sharing. A distributed file system allows a file to be transparent accessed by processes of any node of the system irrespective of the file's location. Therefore, a process on one node can create a file that can then be accessed at a later time by some other process running on another node.

ii) User mobility. In a distributed system, user mobility implies that a user should not be forced to work on a specific node but should have the flexibility to work on different nodes at different times.

iii) Availability. For better fault tolerance, files should be available for use even in the event of temporary failure of one and more nodes of the system. To take care of this, a distributed file system normally keeps multiple copies of a file on different nodes of the system. Each copy is called a replica of the file.

iv) Diskless workstations. Disk drives are relatively expensive compared to the cost of most other parts in a workstation. A distributed file system, with its transparent remote file-accessing capability, allows the use of diskless workstations in a system.

A distributed file system typically provides the following three types of services.

i) Storage service. It deals with the allocation and management of space on a secondary storage device that is used for storage of files in the file system. It provides a logical view of the storage system by providing operations for storing and retrieving data in them.

ii) True file service. It is concerned with the operations on individual files, such as operations for accessing and modifying the data in files and for creating and deleting files.

iii) Name service. It provides a mapping between text names for files and references to files, that is, file IDs. Text names are required because, file IDs are awkward and difficult for human users to remember and use. Most file systems use directories to perform this mapping. Therefore, the name service is also known as a directory service. The directory service is responsible for performing directory-related activities such as creation and deletion of directories, adding a new file to a directory, deleting a file from a directory, changing the name of a file, moving a file from one directory to another, and so on.

**DESIRABLE FEATURES OF A GOOD DISTRIBUTED FILE SYSTEM**

**Q.1 Explain Desirable Features of a Good Distributed File System.**

**(A)** A good distributed file system should have the features described below.

1. **Transparency.** The following four types of transparencies are desirable:
   Structure transparency, although not necessary, for performance, scalability and reliability reasons, a distributed file system normally uses multiple file servers. Each file server is normally a user process or sometimes a kernel process that is responsible for controlling a set of secondary storage devices (used for file storage) of the node on which it runs.

In multiple file servers, the multiplicity of file servers should be transparent to the clients of a distributed file system. In particular, clients should not know the number or location of the file servers and the storage devices. Ideally, a distributed file system should look to its clients like a conventional file system offered by a centralized, time-sharing operating system.

- **Access transparency**. Both local and remote files should be accessible in the same way. That is, the file system interface should not distinguish between local and remote files, and the file system should automatically locate an accessed file and arrange for the transport of data to the client's site.
- **Naming transparency**. The name of a file should give no hint as to where the file is located. Furthermore, a file should be allowed to move from one node to another in a distributed system without having to change the name of the file.
- **Replication transparency**. If a file is replicated on multiple nodes, both the existence of multiple copies and their locations should be hidden from the clients.

2. **User mobility.** In a distributed system, a user should not be forced to work on a specific node but should have the flexibility to work on different nodes at different times. One way to support user mobility is to automatically bring a user's environment (e.g., user's home directory) at the time of login to the node where the user logs in.

3. **Performance.** The performance of a file system is usually measured as the average amount of time needed to satisfy client requests. In centralized file systems, this time includes the time for accessing the secondary storage device on which the file is stored and the CPU processing time. In a distributed file system, however, this time also includes network communication overhead when the accessed file is remote.

4. **Simplicity and ease of use.** Several issues influence the simplicity and ease of use of a distributed file system. The most important issue is that the semantics of the distributed file system should be easy to understand.
   Another important issue for ease of use is that the file system should be able to support the whole range of applications.

5. **Scalability.** It is inevitable that a distributed system will grow with time since expanding the network by adding new machines or interconnecting two networks together is commonplace. Therefore, a good distributed file system should be designed to easily cope with the growth of nodes and users in the system.

6. **High availability.** A distributed file system should continue to function even when partial failures occur due to the failure of one or more components, such as a communication link failure, a machine failure, or a storage device crash.
   Replication of files at multiple servers is the primary mechanism for providing high availability.

7. **High reliability.** In a good distributed file system, the probability of loss of stored data should be minimized as far as practicable. That is, users should not feel

compelled to make backup copies of their files because of the unreliability of the system. Rather, the file system should automatically generate backup copies of critical files that can be used in the event of loss of the original ones.

8. **Data integrity.** A file is often shared by multiple users. For a shared file, the file system must guarantee the integrity of data stored in it. That is, 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. Atomic transactions are a high-level concurrency control mechanism often provided to the users by a file system for data integrity.

9. **Security.** A distributed file system should be secure so that its users can be confident of the privacy of their data. Necessary security mechanisms must be implemented to protect information stored in a file system against unauthorized access. Furthermore, passing rights to access a file should be performed safely; that is, the receiver of rights should not be able to pass them further if he or she is not allowed to do that.

10. **Heterogeneity.** Heterogeneous distributed systems provide the flexibility to their users to use different computer platforms for different applications. Another heterogeneity issue in file systems is the ability to accommodate several different storage media. Therefore, a distributed file system should be designed to allow the integration of a new type of workstation or storage media in a relatively simple manner.

**Video**

**FILE MODELS**

**Q.2 Explain File Models.**

**(A)** ➢ **Unstructured and Structured Files**

According to the simplest model, a file is an unstructured sequence of data. In this model, there is no substructure known to the file server and the contents of each file of the file system appears to the file server as an uninterpreted sequence of bytes. The operating system is not interested in the information stored in the files. Hence, the interpretation of the meaning and structure of the data stored in the files are entirely up to the application programs UNIX and MS-DOS use this file model.

Another file model that is rarely used nowadays is the structured file model. In this model, a file appears to the file server as an ordered sequence of records. Records of different files of the same file system can be of different size. Therefore, many types of files exist in a file system, each having different properties. In this model, a record is the smallest unit of file data that can be accessed, and the file system read or write operations are carried out on a set of records.

Structured files are again of two types- files with nonindexed records and files with indexed records. In the former model, a file record is accessed by specifying its position within the file, for example, the fifth record from the beginning of the file or the second record from the end of the file. In the latter model, records have one or more key fields and can be addressed by specifying the values of the key fields. In file systems that allow indexed records, a file is maintained as a B-tree or other suitable data structure or a hash table is used to locate records quickly.

Most modern operating systems use the unstructured file model. This is mainly because sharing of a file by different applications is easier with the unstructured file model as compared to the structured file model.

In addition to data items, files also normally have attributes. A file's attributes are information describing that file. For example, typical attributes of a file may contain information such as owner, size, access permissions, date of creation, date of last modification, and date of last access.

➢ **Mutable and Immutable files**

According to the modifiability criteria, files are of two types- mutable and immutable. Most existing operating systems use the mutable file model. In this model, an update performed on a file overwrites on its old contents to produce the new contents. That is, a file is represented as a single stored sequence that is altered by each update operation.

On the other hand, some more recent file systems, such as the Cedar File System (CFS), use the immutable file model. In this model, a file cannot be modified once it has been created except to be deleted. The file versioning approach is normally used to implement file updates, and each file is represented by a history of immutable versions. That is, rather than updating the same file, a new version of the file is created each time a change is made to the file contents and the old version is retained unchanged. In practice, the use of storage space may be reduced by keeping only a record of the differences between the old and new versions rather than creating the entire file once again.

**FILE-ACCESSING MODELS**

**Q.3 Discuss File-Accessing Models.**

**(A)** The manner in which a client's request to access a file is serviced depends on the file-accessing model used by the file system. The file-accessing model of a distributed file system mainly depends on two factors – the method used for accessing remote files and the unit of data access.

- **Accessing Remote Files**
  - **i) Remote service model.** In this model, the processing of the client's request is performed at the server's node. That is, the client's request for

file access is delivered to the server, the server machine performs the access request, and finally the result is forwarded back to the client. The access requests from the client and the server replies for the client are transferred across the network as message.

ii) **Data-caching model.** In the remote service model, every remote file access request results in network traffic. The data-caching model attempts to reduce the amount of networks traffic by taking advantage of the locality feature found in file accesses. In this model, if the data needed to satisfy the client's access request is not present locally, it is copied from the server's node to the client's node and is cached there. The client's request is processed on the client's node itself by using the cached data. Recently accessed data are retained in the cache for some time so that repeated accesses to the same data can be handled locally. A replacement policy, such as the least recently used (LRU), is used to keep the cache size bounded.

As compared to the remote service model, the data-caching model offers the possibility of increased performance and greater system scalability because it reduces network traffic, contention for the network, and contention for the file servers.

Many implementations can be thought of as a hybrid of the remote service and the data-caching models. For example, LOCUS and the Network File System (NFS) use. On the other hand, Sprite uses the data-caching model but employs the remote service method under certain circumstances.

- **Unit of Data Transfer**
  - **File-level transfer model.** In this model, when an operation requires file data to be transferred across the network in either direction between a client and a server, the whole file is moved.

**Advantages:**

i) Transmitting an entire file in response to a single request is more efficient than transmitting it page by page in response to several requests because the network protocol overhead is required only once.
ii) It has better scalability because it requires fewer accesses to file servers, resulting in reduced server load and network traffic.
iii) Disk access routines on the servers can be better optimized if it is known that requests are always for entire files rather than for random disk blocks.
iv) Once an entire file is cached at a client's site, it becomes immune to server and network failures. It also simplifies the task of supporting heterogeneous workstations. This is because it is easier to transform an entire file at one time from the form compatible with the file system of server workstation to the form compatible with the file system of the client workstation of vice versa.

**Disadvantages:**

i) The main drawback of this model is that it requires sufficient storage space on the client's node for storing all the required files in their entirety.

ii) If only a small fraction of a file is needed, moving the whole file is wasteful.
**Example –** Amoeba and the Andrew File System.

- **Block-level transfer model.** In this model, file data transfers across the network between a client and a server take place in units of file blocks. A file block is a contiguous portion of a file and is usually fixed in length. For file systems in which block size is equal to virtual memory page size, this model is also called a page-level transfer model.

**Advantages**

i) The advantage of this model is that it does not require client nodes to have large storage space.
ii) It also eliminates the need to copy an entire file when only a small portion of the file data is needed.

**Disadvantages:**

i) An entire file is to be accessed, multiple server requests are needed in this model, resulting in more network traffic and more network protocol overhead. Therefore, this model has poor performance.
**Example** – The Apollo file system, Sun Microsystem's NFS.

- **Byte-level transfer model.** In this model, file data transfers across the network between a client and a server take place in units of bytes.

**Advantages:**

i) This model provides maximum flexibility.

**Disadvantages:**

i) The main drawback of this model is the difficulty in cache management.
**Example** – The Cambridge File Server.

- **Record-level transfer model.** The three file data transfer models described above are commonly used with unstructured file models. The record-level transfer model is suitable for use with those file models in which file contents are structured in the form of records. In this model, file data transfers across the network between a client and a server take place in units of records.

  **Example –** The Research Storage System (RSS).

## FILE-SHARING SEMANTICS

A shared file may be simultaneously accessed by multiple users. In such a situation, an important design issue for any file system is to clearly define when modifications of file data made by a user are observable by other users. This is defined by the type of file-sharing semantics adopted by a file system.

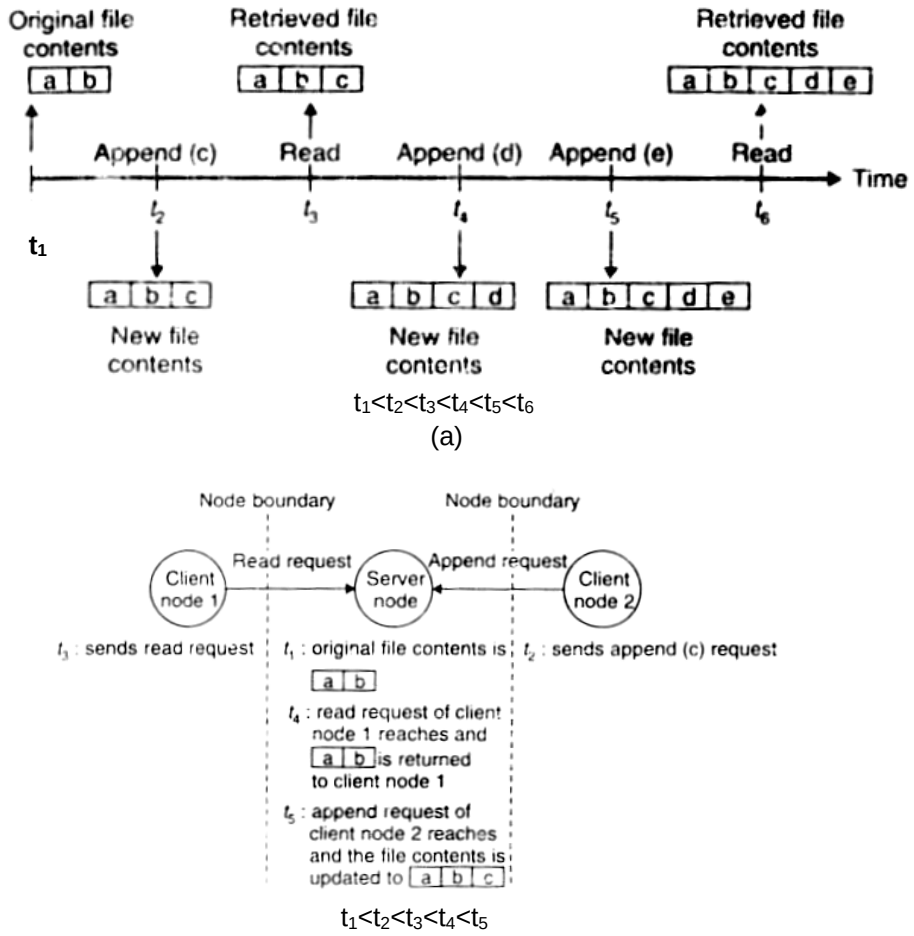



**Fig. 1 :** (a) Example of UNIX file-sharing semantics, (b) It is difficult to achieve UNIX semantics in a distributed file system even when the shared file is handled by a single server.

- **UNIX semantics.** This semantics enforces an absolute time ordering on all operations and ensures that every read operation on a file sees the effects of all previous write operations performed on that file (Figure 1(a)). In particular, writes to an open file by a user immediately become visible to other users who have file open at the same time.

- **Session semantics.** For this semantics, the following file access pattern is assumed: A client opens a file, performs a series of read/write operations on the file, and finally closes the file when he or she is done with the file. A session is a series of file accesses made between the open and close operations. In session semantics, all

changes made to a file during a session are initially made visible only to the client process (or possibly to all processes on the client node) that opened the session and are invisible to other remote processes who have the same file open simultaneously. Once the session is closed, the changes made to the file are made visible to remote processes only in later starting sessions. Already open instances of the file do not reflect these changes.

- **Immutable shared-files semantics.** This semantics is based on the use of the immutable file model. Recall that an immutable file cannot be modified once it has been created. According to this semantics, once the creator of a file declares it to be sharable, the file is treated as immutable, so that it cannot be modified any more. Changes to the file are handled by creating a new updated version of the file. Each version of the file is treated as an entirely new file. Therefore, the semantics allows files to be shared only in the read-only mode. With this approach, since shared files cannot be changed at all, the problem of when to make the changes made to a file by a user visible to other users simply disappears.

- **Transaction-like semantics.** This semantics is based on the transaction mechanism, which is a high-level mechanism for controlling concurrent access to shared, mutable data. A transaction is a set of operations enclosed in-between a pair of begin_transaction and end_transaction-like operations. The transaction mechanism ensures that the partial modifications made to the shared data by a transaction will not be visible to other concurrently executing transaction until the transaction ends (its end_transaction is executed). Therefore, in multiple concurrent transactions operating on a file, the final file content will be the same as if all the transactions were run in some sequential order.

**FILE-CACHING SCHEMES**

**Q.4 How to Perform File Caching ?**

**(A)** In implementing a file-caching scheme for a centralized file system, one has to make several key decisions, such as the granularity of cached data (large versus small), cache size (large versus small, fixed versus dynamically changing), and the replacement policy. A good summary of these design issues is presented. In addition to these issues, a file-caching scheme for a distributed file system should also address the following key decisions:

(a) Cache location (b) Modification propagation (c) Cache validation

These three design issues are described below. :

**(a) Cache Location**

- **Server's main memory.** When no caching scheme is used, before a remote client can access a file, the file must first be transferred from the server's disk to the server's main memory and then across the network from the server's main memory to the client's main memory. Therefore, the total cost involved is one disk access and one network access. A cache located in the server's main memory eliminates the disk access cost on a cache hit, resulting in a considerable performance gain as compared to no caching.
  However, having the cache in the server's main memory involves a network access for each file access operation by a remote client and processing of the access request by the server. Therefore, it does not eliminate the network access cost and does not contribute to the scalability and reliability of the distributed file system.

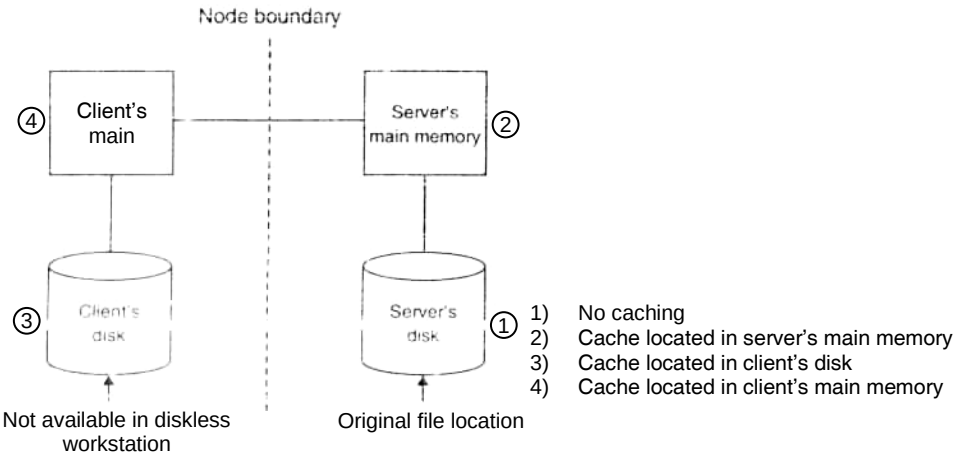



**Fig. 1 :** Possible cache locations in a file–catching scheme for a distributed

- **Client's disk.** The second option is to have the cache in a client's disk. A cache located in a client's disk eliminates network access cost but requires disk access cost on a cache hit.

  **Advantages**

  i) The first is reliability. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover, if the cached data is kept on the client's disk, the data is still there during recovery and there is no need to fetch it again from the server's node.
  ii) The second advantage is large storage capacity. Therefore, more data can be cached, resulting in a higher hit ratio.
  iii) The third advantage is disconnected operation.

  **Disadvantages**

  i) The main drawback of having cached data on a client's disk is that this policy does not work if the system is to support diskless workstations.
  ii) Furthermore, with this caching policy, a disk access is required for each access request even when there is cache hit. Therefore, the access time is still considerably large.

- **Client's main memory.** The third alternative is to have the cache in a client's main memory. A cache located in a client's main memory eliminates both network access cost and disk access cost. Therefore, it provides maximum performance gain on a cache hit. It also permits workstations to be diskless. Like a client's disk cache, a client's main-memory cache also contributes to scalability and reliability because on a cache hit the access request can be serviced locally without the need to contact the server. However, a client's main-memory cache is not

preferable to a client's disk cache when large cache size and increased reliability of cached data are desired.

Cost of remote access in case of no caching
= one disk access + one network access

| Cache location | Access cost on cache hit | Advantages |
|---|---|---|
| Server's main memory | One network access | 1. Easy to implement.<br>2. Totally transparent to the clients<br>3. Easy to keep the original file and cached data consistent.<br>4. Easy to support UNIX-like file-sharing semantics |
| Client's disk | One disk access | 1. Reliabilityagainst crashes.<br>2. Large storage capacity.<br>3. Suitable for supporting disconnected operation.<br>4. Contributes to scalability and reliability. |
| Client's main memory | ___ | 1. Maximum performance gain.<br>2. Permits workstation to be diskless.<br>3. Contributes to scalability and reliability. |

**Fig. 2:** Summary of the relative advantages of the three cache location policies.

**(b) Modification propagation**

In file system in which the cache is located on client's nodes, a file's data may simultaneously be cached on multiple nodes. In such a situation, when the caches of all these nodes contain exactly the same copies of the file data, we say that the caches are consistent. It is possible for the 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.

Keeping file data cached at multiple client nodes consistent is an important design issue in those distributed file systems that use client caching. A variety of approaches to handle this issue have been proposed and implemented. These approaches depend on the schemes used for the following cache design issues for distributed file systems:

i) When to propagate modifications made to a cached data to the corresponding file server.
ii) How to verify the validity of cached data.

**Write-through Scheme**

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

**Advantages**

i) This scheme has two main advantages-high degree of reliability and suitability for UNIX-like semantics. Since every modification is immediately propagated to the server having the master copy of the file, the risk of updated data getting lost (when a client crashes) is very low.

**Disadvantages**

i) A major drawback of this scheme is its poor write performance. This is because each write access has to wait until the information is written to the master copy of the server.

**Delayed-Write Scheme**

Although the write–through scheme helps on reads, it does not help in reducing the network traffic for writes. Therefore, to reduce network traffic for writes as well, some systems use the delayed-write scheme. In this scheme, when a cache entry is modified, the new value is written only to the cache and the client just makes a note that the cache entry has been updated. Some time later, all updated cache entries corresponding to a file are gathered together and sent to the server at a time.

Depending on when the modifications are sent to the file server, delayed-write policies are of different types. Three commonly used approaches are as follows:

i) Write on ejection from cache. In this method, modified data in a cache entry is sent to the server when the cache replacement policy has decided to eject it from the client's cache.
ii) Periodic write. In this method, the cache is scanned periodically, at regular intervals, and any cached data that have been modified since the last scan are sent to the server.
iii) Write on close. In this method, the modifications made to a cached data by a client are sent to the server when the corresponding file is closed by the client.

**Advantages**

The delayed-write policy helps in performance improvement for write accesses due to the following reasons:

i) Write accesses complete more quickly because the new value is written only in the cache of the client performing the write.
ii) Modified data may be deleted before it is time to send them to the server. For example, many programs create temporary files, use them, and then delete them soon after they are created. In such cases, modifications need not be propagated at all to the server, resulting in a major performance gain.
iii) Gathering of all file updates and sending them together to the server is more efficient than sending each update separately.

**Disadvantages**

Delayed-write schemes, however, suffer from reliability problems, since modifications not yet sent to the server from a client's cache will be lost if the client crashes. Another drawback of this approach is that delaying the propagation of modifications to the server results in fuzzier file-sharing semantics, because when another process reads the file, what it gets depends on the timing.

**(c) Cache Validation Schemes**

A file data may simultaneously reside in the cache of multiple nodes. The modification propagation policy only specifies when the master copy of a file at 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. Obviously, a client's cache entry becomes state as soon as some other client modifies the data corresponding to the cache entry in the master

copy of the file. Therefore, 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 basically two approaches to verify the validity of cached data- the client-initiated approach and the server-initiated approach.

- **Client-Initiated Approach**
  i) Checking before every access. This approach defeats the main purpose of caching because the server has to be contacted on every access. But it is suitable for supporting UNIX-like semantics.
  ii) Periodic checking. In this method, a check is initiated every fixed interval of time.
  iii) Check on file open. In this method, a client's cache entry is validated only when the client opens the corresponding file for use. This method is suitable for supporting session semantics.

  The validity check is performed by comparing the time of last modification of the cached version of the data with the server's master copy version. If the two are the same, the cached data is up to data. Otherwise, it is stale and hence the current version of the data is fetched from the server. Instead of using timestamps, version numbers or checksums can be used.

- **Server-Initiated Approach**
  If the frequency of the validity check is high, the client-initiated cache validation approach generates a large amount of network traffic and consumes precious server CPU time. Owing to this reason, the AFS that initially used the client-initiated approach (in AFS-1) switched to the server-initiated approach.

  In this method, a client informs the file server when opening a file, indicating whether the file is being opened for reading, writing, or both. The file server keeps a record of which client has which file open an in what mode. In this manner, the server keeps monitoring the file usage modes being used by different clients and reacts whenever it detects a potential for inconsistency. A potential for inconsistency occurs when two or more clients try to open a file conflicting modes. For example, if a file is open for reading, other clients may be allowed to open it for reading without any problem, but opening it for writing cannot be allowed. Similarly, a new client should not be allowed to open a file in any mode if the file is already open for writing. When a client closes a file, it sends an intimation to the server along with any modifications made to the file. On receiving such an intimation, the server updates its record of which client has which file open in what mode.

  Although the server-initiated approach described above is quite effective, it has the following problems:
  i) It violates the traditional client-server model in which servers simply respond to service request activities initiated by clients. This makes the code for client and server programs irregular and complex.

ii) It requires that file servers be stateful. As explained later, stateful file servers have a distinct disadvantage over stateless file servers in the event of a failure.
iii) A check-on-open, client-initiated cache validation approach must still be used along with the server-initiated approach. For example, a client may open a file, cache it, and then close it after use. Upon opening it again for use, the cache content must be validated because there is a possibility that some other client might have subsequently opened, modified, and closed the file.

In other server-initiated approach, known as the callback policy in AFS a cache entry is assumed to be valid unless otherwise notified by the server.

## 8.2 Distributed File Systems

**Network File System (NFS)**

**Q.1 Explain NFS.**

**(A)** Network file system is developed by Sun Microsystem for their diskless workstations. NFS is a client-server based application layer protocol which allows sharing of directories and files over a network. By using NFS, users and programs can access files on remote systems of non-homogeneous machine as though they were local flies.

**Q.2 State benefits of NFS.**

**(A)**
1) Networked nodes can use less disk space as the data which is common is stored at some other location by everyone and can be accessed by every other client node.
2) No need of separate home directory on every node, only one NFS home directory can be shared.
3) Multiple local copies and physically managing the consistency is avoided.

**Design Goals**

1) To provide machine and operating system independence so that a NFS server can send files to many clients.
2) Easy crash recovery from client-server machine crashes and network issues.
3) The access to remote file should be achieved in the same way as the local file access for a client program.
4) Maintenance of Unix file semantics for Unix clients and the access to remote file should be within reasonable amount of time.

**Q.3 What are NFS services?**

**(A)** The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote file access services. Accordingly, two separate protocols are specified for these services: a mount protocol and a protocol for remote file accesses called the NFS protocol. The protocols are specified as sets of RPCs that define their functionality. These RPCs are the building blocks used to implement transparent remote file access.

The NFS design consists of three major pieces: the NFS protocol, the server side and the client side.

**1) NFS Protocol :** The NFS protocol uses synchronous RPC for communication. The client-server protocol structure of NFS is shown in Figure 1. As the client blocks until the server completes executing, synchronous RPQ emulates in behavior of local procedure call and thus becomes an ideal candidate for NFS implementation above TCP. To ensure faster crash recovery, NFS server is designed as a stateless server which

keeps no information about its clients. This RPC procedure call itself contains all the parameters in its argument to complete the call. When a server crashes, client just keeps resending request until it gets an answer from the server when it boots again.

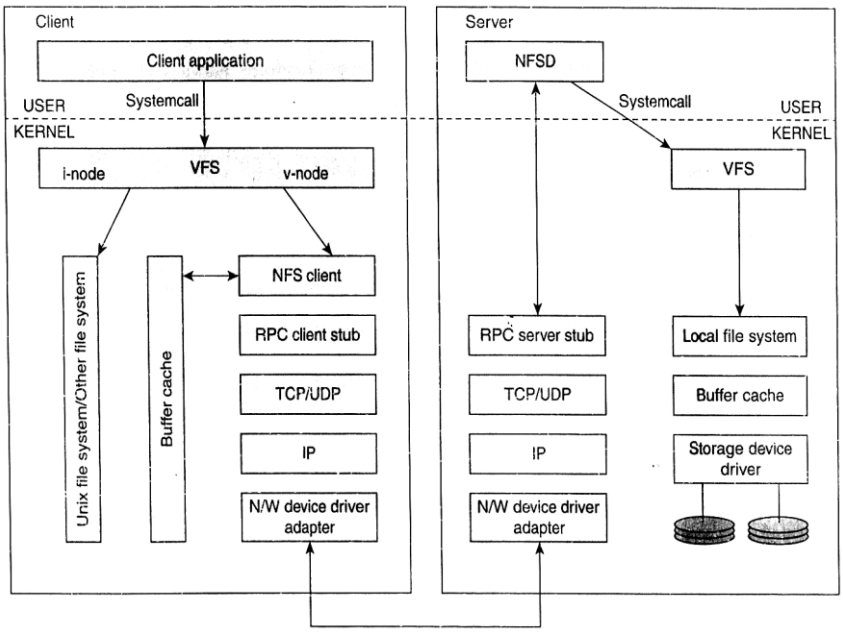



**Fig. 1 :** NFS client-server architecture.

Underneath the RPC layer, the NFS uses TCP/User Datagram Protocol (UDP) and Internet Protocol (IP). UDP is an unreliable datagram protocol and packet delivery guarantee cannot be given through UDP, but due to the stateless server implementation, NFS requests are idempotent, which also helps the client to recover easily. The NFS protocol and RPC use Sun's External Data Representation (XDR) specification for data types specification, making NFS, machine and language independent. The arguments and results of RPC procedures use XDR data definition language.

2) **Server Side :** When a client makes a lookup( ) call for a file residing on the server, the file system returns a file handle, not a file descriptor.
   - **lookup (dirfh, name) returns (fh, attr)**
     This means that the client is asking the filesystem to lookup for a file given by the name "name" and

     **if found (and access rights given) then** return file handle **fh** and attributes of said file in the client's local directory dirfh*/
     else
     return "Fail" if client has no right to access directory dirfh
     fhandle/fh in actual specifies unique location of file which consists of three parts:

- **Filesystem id** identifying the disk partition.
- **I-node number** identifying the exact file within this disk partition.
- **Generation number** changes every time i-node is reused to store a new file.

Server stores **Filesystem id** in filesystem superblock and i-node **generation number** in i-node. There could be a possibility that the server generates a filehandle with an i-node number which is later deleted or removed. This i-node is reused for some other file. Meanwhile the earlier file with the handle is being looked up. The server should be able to figure out that the i-node no. which is now being looked up has been reassigned and the file is a different file. Thus, generation no. is incremented every time and i-node is freed.

The server runs a mount service process running at the user level which lists out all the names of the local filesystems that are available for remote mounting. A known directory, /etc/exports, contains this information along with the access rights for each of the hosts.

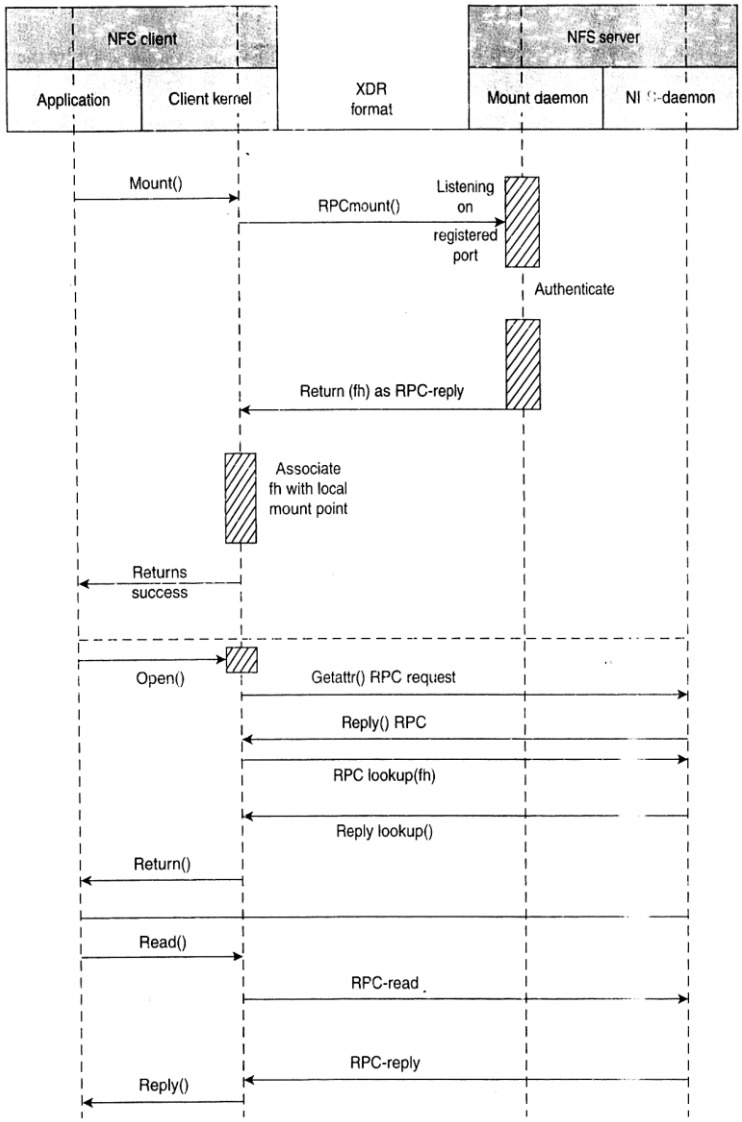



**Fig. 2 :** NFS client-server interaction using protocol procedures.

3. **Client Side :** Client makes a request for a remote directory using the mount command through mount service as shown in above Figure 2. This RPC call has the remote host's name, the pathname of a directory of the remote filesystem and the local name. The remote directory can be any subtree of the remote filesystem; thus client can mount any part of the remote filesystem. The IP address and port number of the server and the file handle are passed on to the VFS layer at the NFS client.

   The VFS (virtual file system) component has a VFS structure for each of the mounted file system and a vnode structure of every open file. The vnode contains a flag to indicate if the file is local or remote. A remote file is accessed and read as shown in above Figure 2. A local file can be accessed

using the reference to the i-node and if remote, it would have the file handle. Once the file is located, transferring blocks of files from the server and caching the blocks in the local memory "buffer cache" requires to be handled.

**Q.4 Explain NFS Caching.**

**(A)** The technique of keeping the data close (Caching the data in the memory), till it is invalidated, is needed to increase the performance of the file system so that the difference in access time between local and remote file is not visible to the client application. Also consecutive updates to the file can be flushed out in a single operation. In NFS, the data is cached from the server to local memory of client, thus avoiding to and from network access to remote disk. As NFS allows multiple client applications to access the file which also may use caching technique, the issues of guarantee of data consistency when multiple processes are reading or writing the same file require to be handled carefully.

**File Attribute Caching**

NFS filesystem caches the file attributes such as length, owner, modification time, as a result of read, write, lookup and readdir commands. This prevents every getattr operation to make a connection to the NFS server. These cached attributes, if not modified, have to be flushed out of the cache every, say, 60 seconds. If modified, they are written back to server immediately.

An NFS client requires to maintain cache consistency by performing cache validity check against the copy of the file on the server. The client uses the attribute, file's modification time. If the cached data time is newer than the modification time of the file then the cached copy of the client remains valid. As soon as the file's modification time becomes newer than the time at which the read data occurred, the cached data must be flushed.

**Client Data Caching**

The NFS client reads the file in the buffer as multiple of the local filesystem block size. This reduces the overhead of RPC read for single byte of data. If the file size is less than the buffer size, the consecutive reads will be from local buffer, avoiding RPC read completely. If the application reads the data not in the buffer, next read fetches the full NFS buffer worth of file data from the remote server. The small writes to the same file are not immediately written back to the server; they are buffered until they are of the size of the full buffer.
Though the above scheme reduces overhead, it introduces problems in presence of client or server crashes.

**Server-Side Caching**

Server-side caching helps in reducing time to access services. There could be three types of server caching policies:

1) The i-node cache maintains the file attributes for ready access as compared to go to disk.
2) As directory searching is expensive, the server has a Directory Name Lookup Cache which contains recently read directories at the time of pathname resolution.

3) Files written to the NFS server cannot be cached (because of the stateless nature of the server); however, buffer cache acts as a read cache for clients.

Note that NFS file locking mechanism allows mutual exclusion to a process requiring access to the file; the other process has to wait for the release of lock. This locking is implemented separately from NFS protocol. The RPC lock deamons for client and servers help in achieving mutual exclusion.

There are three versions or NFS. NFSv2 is older and widely supported. NFSv3 supports 64-bit size files and supports safe asynchronous writes. NFSv4 deprecates the earlier version and does not support rpcbind service and has stateful operations. All versions of NFS can use TCP with NFSv4 mandatorily using it. NFSv2 and NFSv3 can even work with UDP to reduce the protocol overhead for stateless implementation of server. But in case of crashes or lost message with TCP, only the lost frame needs to be resent.

The NFS oilers following services :

1) **Access transparency :** The NFS client application programming interface tor file access whether local or remote is same. No change is required in the programs for remote file access.
2) **Location transparency :** Each client can create its own file namespace which can be of local file-namespace and by mounting remote directories in remote filesystem. The filesystems which are available for mounting are available through mount service running in the server. NFS does not impose any restriction of having system wide namespace. Each client can determine locally the point in its local filesystem where it wants to mount the remote directory.
3) **Scalability :** NFS servers can handle many client requests.
4) **Heterogeneity :** NFS is implemented for most of the known operating system and hardware platforms.
5) **Fault tolerance :** Crash recovery is fast and failure of a client has no effect on any server.
6) **Consistency :** To achieve consistency, validity check from clients is required for each access. This gives one-copy semantics majority of applications. It offers weak consistency.
7) **Security :** Some versions oi NFS use Kerberos.

**VIDEO**

**Q.5 Write short note on Andrew File System (AFS).**

**(A)** The Andrew File System (AFS) was the designed by Professor M. Satyanarayanan of Carnegie-Mellon University in joint effort with IBM. The goal of AFS was to facilitate accessibility of computational and informational facility for campus-wide file system. The choice was to use UNIX semantics.

In NFS the client requires to check with the server if a file has changed or not for consistency validation. This slows down the performance of NFS server in terms of bandwidth and CPU usage. The server can then serve only a few clients as its resources are used to carry out the checking tasks. Also implementing cache consistency which depends on low-level implementation details at the client side is difficult with NFS.

**The File System Design**

The main goal of AFS was to design a shared file system which is scalable in terms of clients supported. The four design decisions of AFS designers took were as follows:

1) AFS should be compatible with UNIX semantics at system call level.
2) Data granularity for movement should be entire file rather than physical or logical record level. This design goal means that the client should copy the entire file to the local disk and all writes have to be written back to the server. The cache at the client side is now able to satisfy its open requests for files. This is known as whole-file-caching strategy of implementation of AFS.
3) Local file caching should be used for reducing the network traffic as well as the server load to enable scalability.

4) The AFS should be designed for graceful growth and availability. Thus, it must use many small servers rather than one large machine.

**Q.6 Discuss implementation of AFS.**

**(A)**

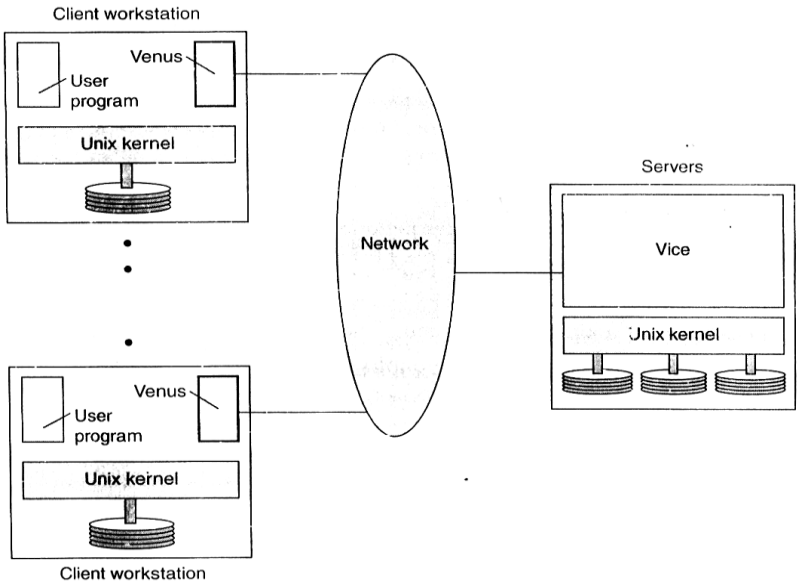



**Fig.:** Distribution of processes in AFS.

The AFS allows the user to access local as well as shared remote files. Local files are stored on a workstations disk and are available only to local user processes. Shared files are stored on AFS servers. For remotely accessing the flies, the AFS implements client-server based R.PC mechanism, where client-side user level process (Venus) communicates with the server-side user level process (Vice). Refer to above Figure.

**Primitives Used**

1) Open( ) : Client system calls for the file (local or remote).If the file being accessed is remote file than the request is passed to Venus.
2) Fetch( ) : Venus sends this RPC request to Vice for the content of the file.
3) Read( ) / Write( ) : Unix Semantics for reading from or writing to the local copy of the file.
4) TestAuth( ) : Test if a file has changed. It is used to check the validity of the cache.
5) GetFileStat: Get the stat info for a file.
6) Store( ) : Store this file on the server.
7) Close( ) : If local copy has changed, Venus sends a copy of the file to the Vice of the file.

**Fetching the File from the Server**

When a client application makes Open( ) system call, the file may not be available locally. This file requires to be now fetched from the remote AFS server, for which the AFS client-side code called Venus sends a Fetch( ) message to the AFS server, requesting for a copy of the file from the server. The fetch call parses the pathname and connects to the file server called the Vice, which traverses the pathname, finds the file in the server file system and sends the copy of the whole file to the client. The fetch call in AFS may also establish callback mechanism. The client-side code caches this file on the local file system (Unix) on the client disk.

An Open( ) call would have used this cached local copy instead of passing the request to Venus if the file would have been available locally and also if its callback promise would have been a valid one (instead of cancelled). An invalid/cancelled callback promise forces the client to fetch the file from the server.

Read( ) and Write( ) system calls are always performed on the locally cached copy of file, without involving communication to the server. Thus the file blocks are cached in the local cache from the local disk.

**File Update to the Server**

When the client application completes working on the file, it is indicated by Close( ). The AFS client directs the server to permanently store this modified version of the file and timestamps with Store( ) message. Note that the local copy on the clients' disk is retained. The next time client wants to use the file it uses the primitive TestAuth( ) to determine if the file has been modified or changed. If not then the client uses the local cached version of the copy rather than the fetching the file from the server. This caching reduces the unnecessary network transfer from the server, but introduces the issues of cache validity and cache concurrency.

**Callbacks: Managing Consistency**

The AFS file system (AFSv2) manages file consistency among the local copies of the various clients using callback mechanism. For every file cached copy at the client, the AFS server maintains callback promise. It is a promise by the server to the clients stating that if there is any update to the file, it would be informed to them. Thus, the clients do not have to contact the server to validate their cache.

When a Fetch( ) is made and the copy of file is sent to the client, Venus places this copy of the file in the local file system of the client, and caches it. It transfers the copy of the file along with its callback promise to the workstation and makes a log of the callback promise. It acts as a token stored with every cached file.

When a server performs any update to the file, following a Close() or a Storc(), it sends callback to all Venus for which it had made callback promises. The local copies set their call promise to cancelled (marking them stale), forcing them to fetch a fresh copy from the server. The file updates to the local copy are immediately visible to all the local processes of the client.

**Concurrency Control**

AFS does not offer support for concurrent updates due to overhead. Any Close() replaces current version on server and all but the update from last close operation at server will be lost. Explicit file locking requires to be used for achieving this.

**Crash Recovery**

When client crashes and reboots, the AFS client sends TestAuth( ) to check if the files which it has have changed or not, as it might have missed the callbacks from the server when it was down. For the file for which the TestAuth is not valid, the client has to fetch the fresh copies of the files from the AFS server.

Server crashes are more complicated as callbacks are kept in memory of the server. Thus, server does not have any idea about the states of the client when it reboots. Each client should realize that the server had crashed and depending on the implementation refetch the files.

**Q.7 What are benefits offered by AFS?**

**(A)**

1) Other than the access transparency, location transparency, heterogeneity and crash recovery, AFS offers better scalability and consistency management as compared to NFS.
2) Also the AFS server maintains a location database mapping the volume name to the name server, thus offering replication transparency for read-only replicas.
3) Even for reading small portion of the data, entire file requires to be read in basic AFS architecture. However, AFS v3 allows data transfer in smaller blocks.
4) The callback mechanism of AFS as compared to the timeout mechanism of NFS for validating the client cache increases the performance of the-AFS

**Q.9 Write a short note on Hadoop Distributed File System (HDFS).**

**(A)** Hadoop, called **Apache Hadoop**, is an Apache Software Foundation for scalable, distributed computing. It is usually used to handle large data sets. It provides fast and reliable analysis for structured as well as unstructured data.

The Apache Hadoop project has a number of sub-projects :

1) Hadoop Common : The common utilities supporting Hadoop sub-projects.
2) Hadoop Distributed File System (HDFS) : A distributed file system that is highly fault tolerant and provides high-throughput access to application data. It works on low-cost machines and implementation is in the Java language.
3) Hadoop MapReduce : A software framework for distributed processing of large data sets on compute clusters.
4) Hadoop YARN : A framework for job scheduling and cluster resource management.

**Q.10 What are goals of HDFS?**

**(A)**

1) HDFS should be designed for large data sets, typically of gigabytes to terabytes. Thus, the HDFS needs large aggregate bandwidth for file transfers which reside on not few but few hundred of nodes.
2) A HDFS instance should support huge number of files.
3) As HDFS is distributed on different machines and large number of components and hardware failures keep occurring in a distributed system, HDFS should have fault detection and quick recovery from failure.
4) The other goal of HDFS is increasing rate of data throughput and thus providing streaming access to data sets.
5) The design should support multiple replicas for data reliability and availability.

**Q.11 Discuss architecture of HDFS.**

**(A) 1) Namespace, NameNode and DataNodes :**

HDFS file system is hierarchical in organization. The files are created, removed, moved, replicated, etc. within the directories of this file system. All information about a file is maintained by a NameNode.

NameNode is the node which acts as master in the cluster nodes in a hierarchical organization. It manages the namespace and controls access to a file. The NameNode acts as the arbitrator and repository of HDFS metadata. The actual file data is stored in the blocks stored in a set of nodes called DataNodes. The NameNodes keep the mapping of block to DataNode. NameNodes execute the system calls like open and close and DataNodes actually serve the clients.

A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine, as shown in Figure.

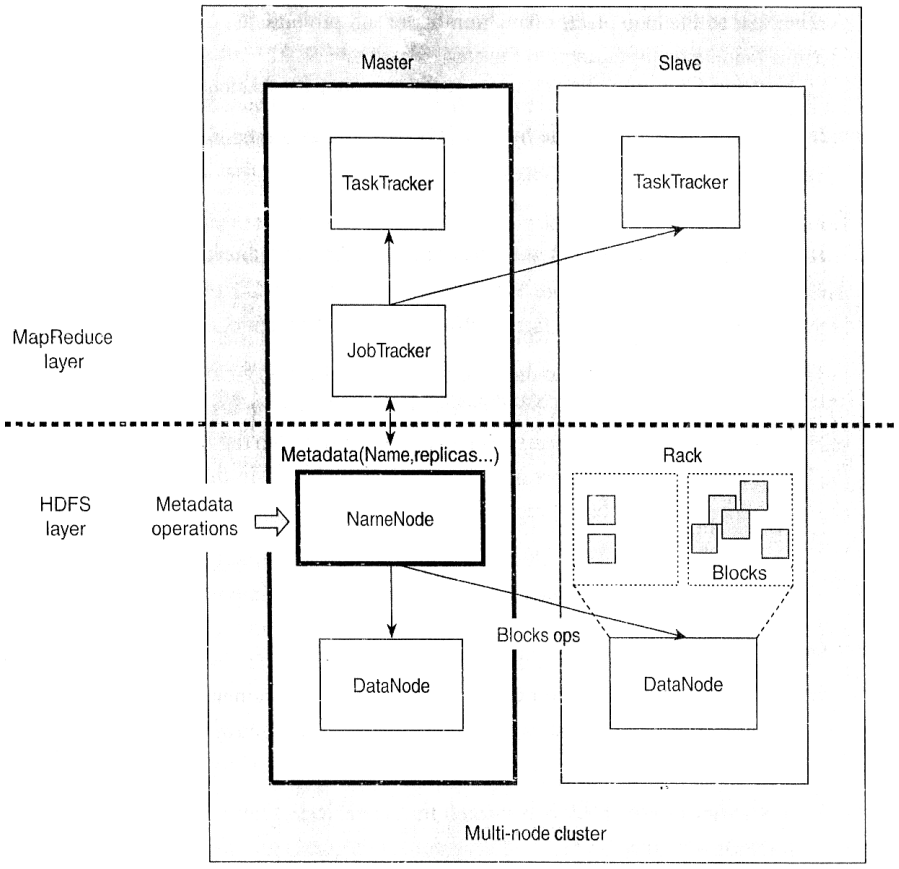



**Fig.:** HDFS architecture for multi-node cluster.

### 2) Data Organization

**Q.12 How data is organized in HDFS?**

**(A)** **Data Blocks:** HDFS is designed to support very large files. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but they read it one or more times and require these reads to be satisfied at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is 64 MB. Thus, an HDFS file is chopped up into 64 MB chunks, and if possible, each chunk will reside on a different DataNode.

A Create( ) request from a client is first cached by HDFS at local file. This local copy, when has data of about HDFS block size, the NameNode is then contacted which in turn allocates a datablock on a particular DataNode. The response of create message is the identifier (ID) of DataNode. If the client has suggested the replication factor to be three, then a list of three DataNodes which will host the replicas are fetched from the NameNode. The client flushes the data block to the first DataNode which pipelines the contents to the next DataNode and the second DataNode pipelines it to the third DataNode.

A file which is deleted is moved to /trash directory and not deleted immediately. This is support file recovery. Only after its expiry time, its data blocks are freed from the DataNode and its entry gets deleted from HDFS namespace.

### 3) Data Replication

HDFS implements replication to support reliability and availability for the large files across machines which are stored in large cluster, possibly spread across many racks. These files are stored as sequence of blocks on the DataNode. With each block, the NameNode keeps a value called replication factor which it constantly monitors.

The blocks are replicated and the block size and replication factor can be decided by the user. The NameNode which has the metadata about the file also stores this information. It periodically (through Heartbeat messages) receives a Blockreport from the DataNode. This report has all the information, such as number of replicas about the block. In the safe mode, the NameNode checks whether the number of replicas are within the minimum limit. If not, the NameNode exits the safe mode and initiates replication of the block on other DataNode.

HDFS implements rack-aware replica placement policy to improve data reliability, availability and network bandwidth utilization. The NameNode determines the rack id of each DataNode via this placement policy which should be optimal in cost and also in bandwidth usage. For example, if replicas are placed on different racks to handle rack failure, it may increase cost of writes due to block transfer needed to multiple racks. General replication factor chosen is three. HDFS uses the nearest replica (the one which minimizes the read latency) for its read request.

If the Heartbeat message does not reach the NameNode, that is if the DataNode has crashed, the DataNode is marked as dead and no read write requests get forwarded to it. The NameNode redirects these requests to other replicated copies on some other DataNode. Any dead node will reduce this factor and if the value goes below a certain threshold the NameNode initializes the process of replication or even re-replication.

Along with replication, HDFS also does data rebalancing and moves data from one block of one DataNode to another block may be on another DataNode. This may be the case when the free space of a DataNode is not up to the desired level or if there is sudden demand for data near client. This is termed as Cluster Rebalancing.

**4) Data Integrity**

The HDFS client implements checksum of each block on the contents of HDFS files and stores it in a hidden file in the same HDFS namespace. The client requires to verify that the data received from DataNode should match the checksum. This ensures integrity of data which is being accessed.

**MapReduce**

Hadoop MapReduce is a software framework for writing applications for processing multi-terabyte data sets, in-parallel on large clusters of commodity hardware in fault-tolerant way. A MapReduce job first maps tasks in parallel manner by splitting input data set into independent sets. The output of this phase then is sorted by the framework and sent to the reduce tasks. The Map Reduce framework schedules, monitors and re-executes in case of failure of task.

Typically MapReduce framework and the Hadoop Distributed File System run on the same set of nodes. Thus the data is readily available and scheduling is faster. This results in high aggregated bandwidth.

The cluster has a single master JobTracker for scheduling, monitoring jobs on each slave whereas for each slave cluster-node, one TaskTracker executes the task allocated to it by master. Refer to Figure.

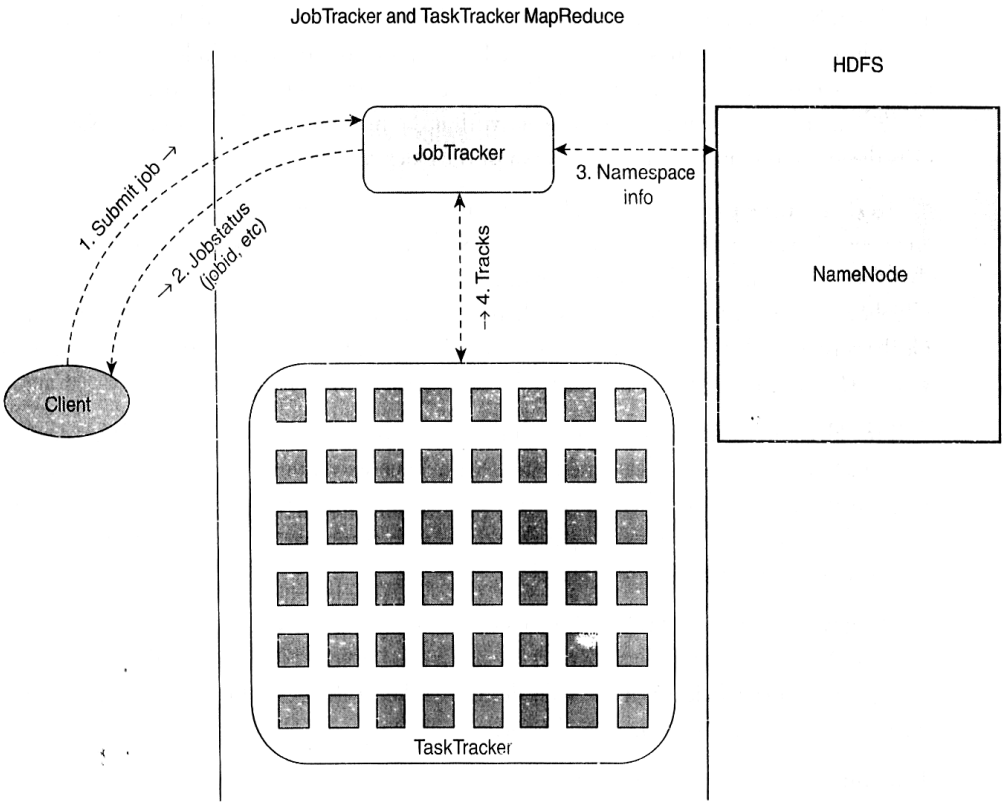



**Fig. :** MapReduce function at Master.

The application specifies the input-output location and map and reduce function by using appropriate interfaces create job configuration. The Hadoop Job client submits the job and the job configuration to JobTracker which distributes the job to slaves. The MapReduce framework views the input as well as the output job as a set of <key, value> pairs.

## References


1. https://www.youtube.com/watch?v=ET7g3L0mfAM
2. https://www.youtube.com/watch?v=fUrKt-AQYtE
3. https://www.youtube.com/watch?v=Lz5DNqfpN1w
4. https://www.youtube.com/watch?v=L_sUyOAjC6M

**Quiz**

**1) What are the different ways in which clients and servers are dispersed across machines ? (Choose Two)**
**a) Servers may run on dedicated machines**
**b) Servers and clients can be on same machines**
**c) Distribution cannot be interposed between a OS and the file system**
**d) OS cannot be distributed with the file system a part of that distribution**

**2) What are the characteristics of a DFS ? (Choose Two)**
**a) login transparency and access transparency**
**b) Files need not contain information about their physical location**
**c) No Multiplicity of users**
**d) No Multiplicity if files**
**Answer : a & b**

**3) What is not a characteristic of a DFS ? (Choose three)**
**a) Fault tolerance**
**b) Scaleability**
**c) Heterogeneity of the system**
**d) Upgradation**
**Answer : a,b & c**

**4) What are the different ways file accesses take place ? (Choose three)**
**a) sequential access**
**b) direct access**
**c) random**
**d) indexed sequential access**

**5) What are the major components of file system ? (Choose three)**
**a) Directory service**
**b) Authorization service**
**c) Shadow service**
**d) System service**
**Answer : a,b & d**

**6) What are the different ways mounting of file system ? (Choose three)**
**a) Boot mounting**
**b) root mounting**
**c) explicit mounting**
**d) auto mounting**
**Answer : a,c & d**

**7) What is the advantage of caching in remote file access ?**
**a) Reduced network traffic by retaining recently accessed disk blocks**
**b) Faster network access.**
**c) Copies of data creates backup automatically**
**d) None of these**

**Answer : a**

**8) What is networked virtual memory ?**
**a) Caching**
**b) Segmentation**
**c) RAM disk**
**d) None of these**
**Answer : a**

**9) What are examples of state information ? (Choose three)**
**a) opened files and their clients**
**b) file descriptors and file handles**
**c) current file position pointers**
**d) current list of total users**
**Answer : a,b & c**

**10) What are examples of state information ? (Choose three)**
**a) Mounting information**
**b) Description of HDD space**
**c) Session keys**
**d) Lock status**
**Answer : a,c & d**

**GQ**

**1.** What does distributed file system normally support?

**2.** What kind of services DFS provides?

**3.** What are the desirable features of a good DFS?

**4.** What are the main approaches to verify the validity of cached data in DFS?

**5.** What are unstructured and structured files?

**6.** Explain mutable and Immutable files.

**7.** Explain in depth file accessing models.

**8.** With neat diagrams, explain the failure handing mechanisms for message passing.

**9.** Give advantages and disadvantages of Block-level, File-level transfer model and Byte-level transfer model.

**10.** Explain File-sharing semantics.

**11.** Explain File-caching schemes.

**12.** Explain File-replication in detail.

**13.** What are the issues in file replication?

**14.** Write short notes :
- (i) Read only replication
- (ii) Available-copies protocol
- (iii) Read-Any-Write All protocol
- (iv) Primary copy protocol
- (v) Quorum- Based protocol

**15.** Differentiate : Replication vs Caching?

**16.** Discuss file caching for distributed system.

**17.** What are the good features of a Distributed File Systems? Explain file sharing sematics of it.

**18.** What do you mean by a Consistency Model? Explain the available consistency models and the requirements of the systems which support them.